Plug: Automatically Tolerating Memory Leaks in C and C++ Applications

Gene Novark Emery D. Berger
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
gnovark@cs.umass.edu, emery@cs.umass.edu

Benjamin G. Zorn
Microsoft Research
One Microsoft Way
Redmond, WA 98052
zorn@microsoft.com

Abstract

Memory leaks remain a significant challenge for C and C++ developers. Leaky applications become slower over time as their working set grows, triggering paging, and can eventually become unresponsive. At the same time, memory leaks remain notoriously difficult to debug, and comprise a large number of reported bugs in mature applications. Existing approaches like conservative garbage collection can only remedy leaks of unreachable objects. In addition, they can impose unacceptable runtime or space overheads, or cause legal C/C++ applications to fail or retain excessive memory.

This paper presents Plug, a runtime system for C/C++ applications that allows applications to deliver high performance in the face of both reachable and unreachable memory leaks. It uses a novel heap layout that isolates leaked objects from non-leaked objects, allowing them to be completely paged out to disk. Plug further reduces the space impact of leaks by employing *virtual compaction*, an approach that leverages virtual memory primitives to allow physical memory compaction without moving objects. We demonstrate Plug's low overhead and its effectiveness at tolerating real memory leaks.

1. Introduction

Memory leaks continue to plague the developers of applications written in C and C++. They continue to be one of the most common types of reported bugs, even for mature projects. For example, in the first two months of 2008, over 150 memory leak bugs were reported in the Firefox browser [27]. While memory debugging tools like Purify [11] and Valgrind [28] can help programmers detect these errors in short-lived applications, these bugs are notoriously difficult to detect and debug in long-running applications like servers or web browsers.

Although memory leaks can eventually lead to memory exhaustion, their primary symptom is performance degradation. Leaks cause an application's working set to grow as the program runs. If the program runs long enough, this increased working set size eventually exceeds available physical memory, triggering paging. The resulting thrashing of pages between main memory and swap space can make applications unresponsive.

While conservative garbage collection [6] has been used to combat memory leaks in deployed C/C++ applications, it is not a complete solution. First, garbage collection can impose significant runtime and memory overhead [14], making it unsuitable for many applications. Second, conservative garbage collection can cause legal C/C++ programs to fail or exhibit unbounded memory consumption [2, 4, 5, 15]. Critically, even when garbage collection does not compromise performance or correctness, it can not resolve all memory leaks. A garbage collector can only reclaim objects whose last reference has been removed. If the objects remain reachable, a garbage collector cannot reclaim them.

This paper makes the following contributions:

1. It presents **Plug**, a runtime system for C/C++ that tolerates memory leaks. Unlike garbage collection, Plug does not impose significant runtime or memory overhead, and allows applications to tolerate leaks regardless of whether the leaked objects are reachable.
2. It introduces two new allocation algorithms that form the basis of Plug: (1) **context-sensitive** memory allocation and (2) **age-segregated** memory allocation.
3. It introduces **virtual compaction**, a memory management algorithm that leverages standard virtual memory primitives to allow the compaction of objects in physical memory while retaining their original locations in virtual memory, making it suitable for C and C++.

We demonstrate Plug's low overhead and effectiveness across a suite of standard benchmarks and applications with real memory leaks. For non allocation-intensive workloads, Plug imposes low runtime (3% across the SPECint benchmark suite) and memory overhead, while effectively eliminating the impact of memory leaks on application working sets (reducing them by up to 55%).

2. Overview

Plug relies on a memory allocator that isolates hot objects from leaked objects so that leaks do not increase the application's working set size. Because C and C++ do not permit object relocation, the only way to separate hot objects from leaked (and thus cold) objects is at allocation time.

Plug's allocator works by segregating objects along two different axes. First, Plug uses a **context-sensitive** allocation strategy: it uses the calling context of `malloc` calls to segregate objects from different *allocation sites* onto different pages. Because memory leaks typically affect objects from a small number of allocation sites, this segregation precludes most objects from ever being allocated on the same page as a future leak.

Plug combines this context-sensitive allocation with an **age-segregated** memory allocator. Age segregation ensures that all objects on the same page are of the same age, as measured by allocation time. Eventually, as the program frees non-leaked objects, leaked objects will be isolated on their own pages. Once these pages become cold, they will get paged out to disk, and never be touched again.

Plug further refines these algorithms to reduce the risk of excessive fragmentation. First, Plug performs per-allocation site segregation only for sites that are the source of a large number of objects. This approach prevents the worst-case of having a page allocated to hold a single small object allocated from each call site. Notice that this policy does not impair Plug's leak tolerance: by definition, sites that allocate few objects cannot be the source of significant memory leaks.

In addition, Plug performs **virtual compaction**, a novel technique that allows the compaction of physical memory without the need to move objects (which C and C++ do not permit). Virtual compaction retains segregation within *virtual* address space, while significantly reducing the *physical* fragmentation of the heap. Thus, virtual compaction can reclaim a substantial amount of physical memory when a small number of live (leaked) objects are spread across a number of pages.

Finally, Plug performs lightweight tracking of reference information to ensure that Plug compacts only pages with cold objects, ensuring that hot objects are not mixed with potential leaks.

While no system can stop leaks from eventually exhausting available address space (especially on 32-bit systems) or available swap space on disk, Plug can prevent leaks from degrading application performance and thus keep applications running longer and more efficiently.

Outline

The rest of this paper is organized as follows. Section 3 describes Plug's segregating allocator in detail. Section 4 describes the virtual compaction mechanism Plug uses to reduce memory overhead. Section 5 describes the limitations of leak tolerance, and of Plug in particular. Section 6 empirically evaluates Plug's runtime and memory overheads and leak toleration. Section 7 presents an overview of related work, Section 8 presents directions for future work, and Section 9 concludes.

3. Plug Heap Structure

Plug tolerates memory leaks by preventing objects that are still in use from sharing pages with leaked objects. If leaked objects share no pages with application data, they cannot increase an application's working set, and thus will not degrade application performance.

While garbage-collected languages like Java support moving garbage collection [16], C's and C++'s direct access to memory addresses precludes object relocation. Thus, the only way to prevent leaked objects from mixing with live objects is to separate them at allocation time.

To achieve this separation, Plug uses a novel memory manager that segregates objects *a priori*. Figure 1 presents an overview of Plug's memory manager, which segregates objects by two dimensions: *allocation sites* (the calling context that ends in `malloc` or `new`) and *age* (in allocation time).

3.1 Allocation-Site Segregation

Previous research has shown that objects allocated from the same call site tend to exhibit similar behavior and lifetime patterns [32]. To isolate leaks, Plug segregates objects by associating a separate heap with each allocation site. Plug identifies these sites with bounded context sensitivity (the last four functions on the call stack).

Each heap then uses a distinct set of pages to satisfy allocation requests from that site. This segregation helps to prevent the intermingling of objects from sites that produce hot objects with those that produce cold or leaked objects.

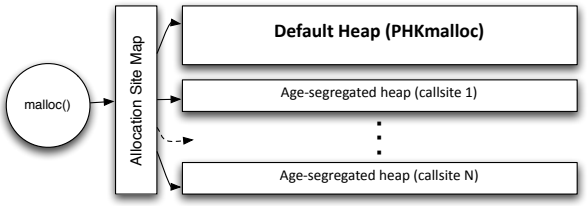


Figure 1. Plug's context-sensitive heap structure, which segregates allocation requests by *allocation site* once the number of live objects exceeds a fixed threshold (see Section 3.1).

```
1  void * plugmalloc (size_t size) {
2    // compute hash of calling context.
3    int context = getContextHash();
4    Metadata * m = getMetadata(context);
5    // one more object allocated.
6    m->liveCount++;
7    // use the age-segregated heap to
8    // satisfy the request, if possible.
9    if (m->getAgeSegHeap() != NULL) {
10     return m->getAgeSegHeap()->malloc (size);
11   } else if (m->getLiveCount() >= 64) {
12     // make a new heap.
13     m->initAgeSegHeap();
14     return m->getAgeSegHeap()->malloc (size);
15   } else {
16     // still below threshold:
17     // get memory from standard allocator.
18     return phkmalloc_with_header (size,
19         context);
20   }
21 }
```

Figure 2. Pseudo-code for Plug's allocation-site segregated `malloc`.

```
1  void plugfree (void * ptr) {
2    // check pointer validity.
3    if (!isFromPlugHeap(ptr)) {
4      int context = getHeader(ptr);
5      Metadata * m = getMetadata(context);
6      m->liveCount--;
7      // return to standard allocator.
8      phkfree_with_header (ptr);
9    } else {
10     void * page = ptr & ~(PAGE_SIZE-1);
11     PageEntry * entry = pageMap(page);
12     entry->free (ptr);
13   }
14 }
```

Figure 3. Pseudo-code for Plug's allocation-site segregated `free`.

The result is that pages tend to fall into two classes: those that contain all cold objects, which can be swapped to disk with little performance penalty, or mostly hot objects, which increases the page-level spatial locality of the heap. Contrast this separation with the behavior of conventional memory allocators, which do not perform per-call site segregation and thus can end up with a single hot object on a page filled with cold or leaked objects.

Limiting Memory Overhead

Most applications have a large number of dynamic allocation sites. For example, Firefox allocates from approximately 14,000 sites during most runs. However, most sites produce relatively few objects, especially those that correspond to Firefox's initialization phase. Allocating an entire page to hold a few small objects wastes memory.

To reduce this memory overhead, Plug instantiates a new heap only for a site when the number of live objects from that site reaches 64. Plug adds an extra header word to each object in its default heap to track the allocation site of each object. When objects are freed, Plug decrements the live count for the object's initial allocation site.

At allocation time, Plug uses a hash table to map each allocation site to a metadata entry (line 4 of Figure 2), which tracks statistics including the total allocation count for the site. As long as the total live count for that site remains below 64, Plug allocates object requests directly from a conventional heap (line 18). Plug uses PHKmalloc [18] as its default allocator. Otherwise, it instantiates a separate heap for that site (line 13), using it for all subsequent allocations from that site. This approach filters out sites that only produce a small number of objects, since these sites cannot be sources of substantial memory leaks.

3.2 Age-Based Segregation

While call site segregation is a heuristic that can help separate objects with similar behavior, it does not guarantee that a call site will only generate objects that are either mostly cold or mostly hot.

Plug departs further from conventional heap layouts by isolating objects by *age*, as measured in allocation time. The key insight is the following: leaked objects by definition are never reclaimed, and thus become older and older as program execution continues. Keeping old objects separate from newer objects thus prevents leaks from intermingling with newer, potentially hot objects.

Plug separates young from old objects in what we call an **age-segregated heap**. Each age-segregated heap is itself a segregated-fits allocator [38] organized as a collection of pages. Each page is an array of fixed-sized object slots (see Figure 4). Each heap contains a list of pages for each size class (powers of two, ranging from 16 to 2048 bytes), plus a special bin for larger objects.

Plug satisfies allocation requests by bumping a pointer through the currently active page for the appropriate size class (line 26 of Figure 5). When an active page is filled, Plug maps a fresh (empty) page and uses it for subsequent allocations (lines 11–18). Free operations decrement the population count for the appropriate page (line 3 of Figure 6). Plug only reuses memory on a page when the population count for a page drops to zero and the bump pointer has reached the end of the page (lines 5–8). If the page less than half live, Plug adds it to the aging queue for virtual compaction, described in Section 4 (lines 9–12).

Plug assigns one metadata structure for each allocated page. This structure contains three elements: (1) the bump pointer, used for allocation from non-full pages; (2) the total number of live objects, which lets Plug free pages when their population drops to zero; and (3) a bitmap that tracks which slots contain live objects, used by Plug's **virtual compaction** algorithm. Plug uses a two-level page table structure to map page addresses to metadata.

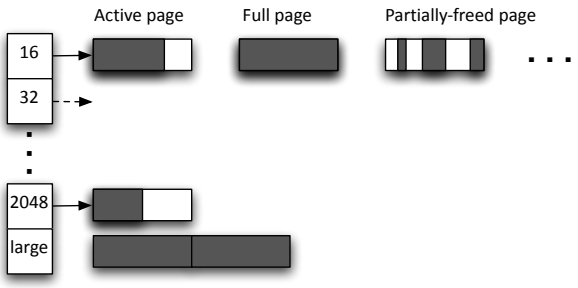


Figure 4. Plug's age-segregated heap (see Section 3.2).

```
1  void * AgeSegHeap::malloc (size_t size) {
2    if(size > PAGE_SIZE/2)
3      return allocWithMmap(size);
4
5    int c = computeSizeClass (size);
6    Heap * h = getHeapFromClass (c);
7
8    if (!h->activePage ||
9      h->activePage->bump
10     == h->activePage->endOfPage) {
11     void * page = getNewPage();
12     PageEntry * e =
13       createPageEntry (page);
14     e->bump = page;
15     e->endOfPage = page + PAGE_SIZE;
16     e->inUse = 0;
17     e->heap = h;
18     h->activePage = e;
19   }
20
21   return h->activePage->malloc();
22 }
23
24 void * PageEntry::malloc(size_t size) {
25   void * ptr = bump;
26   bump += roundUp(size);
27   inUse++;
28   bitmap.set(indexOf(ptr));
29   return ptr;
30 }
```

Figure 5. Pseudo-code for Plug's age-segregated `malloc`.

4. Virtual Compaction

Plug recycles memory from age-segregated heaps only when pages become completely empty. This strategy could potentially lead to to high fragmentation. In the worst case, a single live object could prevent the reclamation of an entire page.

To mitigate this problem, Plug uses a novel scheme we call **virtual compaction** that leverages standard virtual memory remapping primitives to permit compaction of multiple virtual pages onto the same physical page, without moving objects in virtual address space. While we limit our discussion here to its use in Plug, we believe that virtual compaction may enable a new class of compacting memory managers for C and C++ applications.

Virtual compaction merges virtual pages with no overlapping objects into a single physical page. This process is facilitated by Plug's age-segregated heaps, which use a segregated fits structure

```
void PageEntry::free (void * ptr) {
  // find originating page.
  inUse--;
  bitmap.clear(indexOf(ptr));
  if ((inUse == 0) && (bump == NULL)) {
    // free the page for re-use.
    recyclePage (page);
    clearPageEntry (page);
  } else if ((inUse < NUM_ENTRIES/2)
       && (bump == NULL)) {
    // check virtual compaction queue
    AgingQueue.addOrUpdate(this);
  }
}
```

Figure 6. Pseudo-code for Plug's age-segregated `free`.

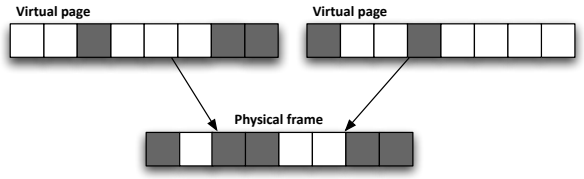


Figure 7. An example pair of pages that share no common live indices. The virtual compactor can merge these pages in physical memory while not actually relocating objects in virtual memory (see Section 4).

in which each page is an array of identically-sized objects. Plug maintains a bitmap per page indicating which indices within the page are occupied by live objects. If a pair of pages contain live objects only at different indices (i.e., there is no index corresponding to a live object on both pages), then the pages can be overlaid on top of each other with no collisions between live objects (see Figure 7). Our current implementation only considers merging pages within the same size class for simplicity. Merging pages with different-sized objects would enable more virtual compaction, but require tracking more metadata.

Using the Linux `mremap` call, Plug merges such pairs of pages onto a single physical frame and maps that frame to the virtual addresses of both original pages. Thus, while virtual memory remains highly fragmented (because virtual memory is only recycled at page-granularity), virtual compaction significantly increases the occupancy of physical pages, reducing the footprint of the application.

Plug tracks stale pages with less than 50% occupancy using a *fragmentation manager*. Prior to being placed in the fragmentation manager, Plug filters out hot pages using an *aging queue*. Pages on the queue are protected, so any access causes a page fault, which Plug uses to update staleness information. Only pages beyond a certain threshold are considered for merging. Figure 8 shows the life cycle of a page as it transitions between states.

Virtual compaction can be implemented in many ways. This section describes when and how Plug identifies pairs of pages to compact, as well as the virtual memory-based mechanism for merging pages.

4.1 Finding Candidate Pairs

At runtime, Plug's heap can contain many low-occupancy pages. Pages in the heap may be modeled as a graph, where each page is a node. Edges exist between two pages when they share a common live object index, and thus cannot be merged via virtual compaction. In this model, finding an *optimal* compaction strategy (fewest physical pages) is equivalent to graph coloring, and thus NP-complete. Plug therefore makes no attempt to optimally compact pages, and instead relies on heuristics that are effective in practice (see Section 6.3).

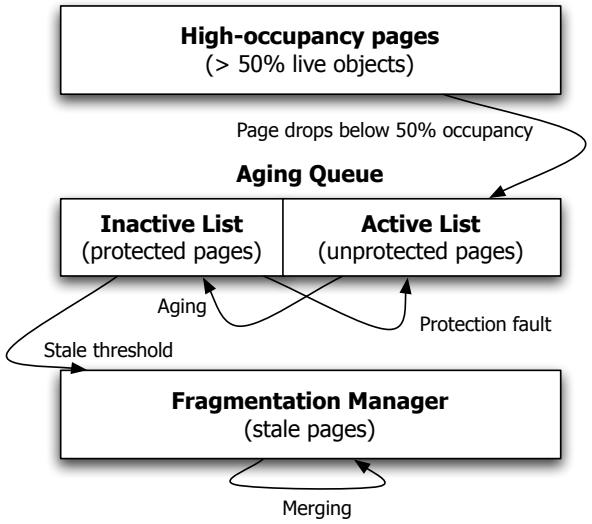


Figure 8. The life cycle of a page in Plug (see Section 4.2.1).

```
// called when a page is added
// to the frag manager or an object
// is freed on the page
void FragManager::checkMerge (PageEntry * p)
{
  for each (PageEntry * q in pageList) {
    if (!p.conflicts(q)) {
      // virtual compact p together with q.
      p.mergeWith(q);
      return;
    }
  }
}
```

Figure 9. Pseudo-code for Plug's first-fit virtual compaction algorithm (see Section 4).

In fact, Plug faces a more difficult problem than ordinary graph coloring, because the graph constantly changes as objects are deallocated. The current prototype uses a simple first-fit strategy to identify pairs of pages to compact.

Plug considers compaction only for pages which have crossed a staleness threshold. Plug moves these pages from the aging queue to the fragmentation manager. Figure 9 shows pseudocode of the fragmentation manager's compaction algorithm. When a deallocation occurs on a page managed by the fragmentation manager, it scans its list to find another page which has no conflicts. If it finds a compatible target, then it eagerly merges the two pages.

Pages can be tested for compatibility quickly by performing a bitwise AND of their live object bitmaps. If any bit in the result is set, then the pages conflict.

```
void inactiveFault (PageEntry * e) {
  faultCount++;
  inactiveList.remove(e);
  activeList.push(e);
}

void activeAdd (PageEntry * e) {
  activeList.push(e);
  if(inactiveList.size() < targetSize) {
    for(i = 1 to 8) {
     inactiveList.push(activeList.pop());
    }
  }
  if(elapsedTime > 125 ms) {
    updateTargetSize();
  }
}

void updateTargetSize() {
  overhead = faultCount * .5 / elapsedTime;
  if(overhead < 0.5%) {
    targetSize +=
      max(min(inactiveList.size,
        activeList.size)/32,8);
  } else if(overhead > 1.5%) {
    targetSize -=
      max(min(inactiveList.size,
        activeList.size)/8,8);
  }
  faultCount = 0;
  elapsedTime = 0;
}
```

Figure 10. Pseudo-code for Plug's segregated aging queue.

4.2 Merging Pages

When merging pages, Plug first iterates through the liveness bitmap of one page, copying the live objects onto the target page. It then remaps the target physical page to *both* virtual addresses. This is done using the `mremap` system call and specifying a size of 0 [36]. The virtual pages thus share a single physical frame, reducing memory overhead.

Virtual compaction is not limited to pairs of pages. Any number of virtual pages may be combined onto a single physical frame as long as they have no conflicts.[1] Thus, merged pages are put back onto the candidate list for further compaction. A merged page contains a bitmap representing the combined live object information for the corresponding virtual pages, enabling fast conflict checking.

4.2.1 Cold Object Filtering

If Plug were to naïvely merge pages, it might accidentally combine hot objects with leaked objects, negating the benefits of segregation. To avoid this problem, Plug applies virtual compaction only to cold pages, those that have been stale for a long time.

Figure 8 illustrates the life cycle of pages in Plug. As the figure shows, before considering a page for virtual compaction, Plug places it on the *aging queue* which is composed of two lists. Plug keeps all pages below 50% occupancy on the queue. A higher threshold would enable more virtual compaction, but high-occupancy pages are much harder to match with merge targets. A lower threshold would reduce the overhead for tracking candidates, but allow less compaction. The aging queue is organized in order of staleness, measured as the time since the application last accessed some object on the page.

To determine staleness, Plug protects the pages on the queue against direct read and write access by the application using the `mprotect` system call. If an object on the page is accessed, the Plug runtime catches the protection fault and unprotects the page. It finally moves the page to the head of the queue for further aging.

Protecting all pages on the queue would be prohibitively expensive, since some pages will contain frequently-used objects. Plug thus segregates the aging queue into *active* and *inactive* lists. Pages on the inactive list are page-protected and managed in LRU order, while pages on the active list are unprotected and managed using FIFO. When the program accesses a page on the inactive list, a page fault occurs, and Plug moves the page to the head of the active list. Plug periodically moves pages from end of the active list onto the inactive list to maintain the latter's target size.

4.3 Adapting the Inactive List

The size of the inactive list is controlled adaptively to achieve both acceptable runtime overhead and to gather useful information on page accesses. Since each page on the inactive list is protected, a larger inactive list gathers more useful data about page staleness, but results in more runtime overhead due to page faults. Plug's heuristics for controlling the list sizes and moving objects from the active to inactive list are based on those used in CRAMM [41].

Each time a page is added to the aging queue, Plug checks whether it should adjust the sizes of the queues. If 1/8 of a second of CPU time has passed (or 10 page faults), Plug reevaluates the sizes. Plug estimates the runtime overhead caused by minor page faults to the inactive list (using an estimate of 500 μs minor page fault cost). If this cost is above 1.5% of total CPU time, Plug decreases the target size of the inactive list. If it is less than 0.5%, Plug increases its size.

Plug maintains a *target inactive size*, initially 0. When changing the size of the inactive list, Plug is actually changing the target. The actual inactive list size will gradually approach the target. This policy prevents a sudden spike in minor page fault overhead by immediately protecting a large number of pages.

Plug's size adjustments are the same as in CRAMM. If the active list currently holds P_A pages and the inactive list P_I, then the new target inactive size will be:

- Increase: $P_I = P_I + \max(\min(P_A, P_I)/32, 8)$
- Decrease: $P_I = P_I - \max(\min(P_A, P_I)/8, 8)$

These adjustments reflect the need to make small adjustments when the lists are small, and larger adjustments when the lists are larger. They also ensure that some minimum adjustment is always made. Plug decreases the target inactive list size more aggressively than it increases it, as the goal of low runtime overhead takes precedence over more accurate information.

The active list always holds all pages in the aging queue that are not in the inactive list. Each time a new page is added to the queue, Plug checks the current inactive list size against the target. If the current list is too small, Plug moves up to 8 pages from the active list to the inactive list. When the target size is smaller than the current size, Plug lazily allows the inactive list to shrink as pages move to the active list due to page faults. This policy ensures that truly inactive pages never move from the inactive to the active list, and thus are guaranteed to cross the staleness threshold and move to the fragmentation manager.

Plug uses the aging queue to identify stale pages which should be virtually compacted. Whenever a new page is added to the aging

[1] Kernel limitations restrict the maximum number to 128, which is not a problem in practice.

queue, Plug examines the stalest pages on its inactive list. It moves those beyond a given threshold of staleness (currently 25,000 heap allocations) to the fragmentation manager for virtual compaction.

While it is possible that a merged page could later become hot again, making it advantageous to resegregate the objects onto separate pages, we have not observed this phenomenon in practice.

5. Discussion

While Plug provides significant advantages in tolerating memory leaks, it also has limitations. As noted, Plug reduces the memory footprint of leaky applications, but ultimately cannot reduce the address space requirements of these applications. In a 32-bit address space, many leaking applications will eventually run out of address space using any existing approach to reducing the impact of leaks, including conservative garbage collection. Fortunately, multi-gigabyte physical memories have led computer vendors to move aggressively to 64-bit systems, where address space limitations are far less problematic. Similarly, most storage systems provide ample room for swapped out pages.

A broad concern related to systems like Plug, which reduce the impact of errors in existing programs, is that they could lead to sloppy programming and encourage the continued use of weakly-typed languages like as C and C++. While type-safe languages offer significant software engineering advantages over their unsafe counterparts, addressing problems in C and C++ applications is significant both because of the large body of deployed code in these languages, and the fact that C and C++ remain the languages of choice for many important application domains. Nevertheless, Plug is not a panacea for mistakes in explicit deallocation, as it provides no protection against dangling pointers. However, one benefit of the leak tolerance that Plug provides is that it might reduce the likelihood that programmers will over-aggressively free objects, turning a performance error into a more serious memory corruption error.

While Plug attempts to reduce the overhead of its mechanisms for segregating leaking objects, problematic object behavior can reduce its effectiveness. For example, if cold objects that have become segregated become hot again (perhaps because the user returns to using some functionality after a long hiatus), our current implementation does not re-segregate them. While there are no technical reasons this could not be done, we leave a consideration of the policies that would be needed for future work.

Finally, in the worst case, it is possible for a leaky application to defeat Plug's strategy of segregating by allocation site and age. For example, an application that repeatedly allocates hot objects with leaks at roughly the same time and the same size and allocation site will render Plug's segregation ineffective. However, this scenario will only arise when the application is exhibiting catastrophic leak behavior, which is generally easy to debug before deployment.

6. Experimental Results

We first evaluate Plug's performance overhead on a suite of benchmark applications. We then evaluate its efficacy at tolerating leaks by reducing working set size.

6.1 Runtime Overhead

We evaluate Plug's performance on the SPECint2000 benchmarks [35] on their reference workloads[2], as well as with a suite of allocation-intensive benchmarks. The latter benchmarks stress the memory allocator due to high allocation rates and have been widely used in memory management studies.

[2] `252.eon` fails to run under both Plug and GNU libc, and `253.perlbmk` fails to run with our current implementation of Plug.

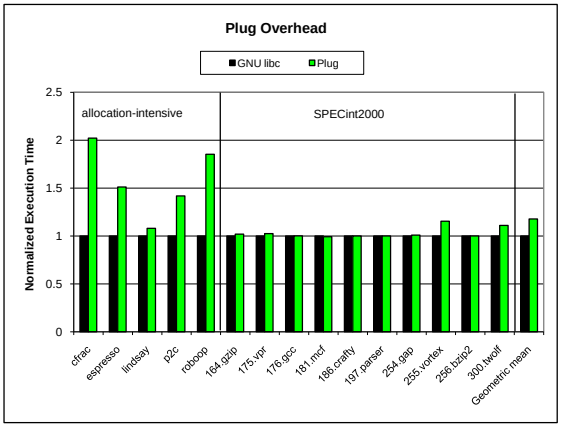


Figure 11. Runtime overhead for Plug across a suite of benchmarks, normalized to the performance of the GNU libc (Linux) allocator (see Section 6.1). Plug's overhead is substantial for highly allocation-intensive benchmarks, but modest (3.0%) for SPECint.

Our experimental machine is a single-core, hyperthreaded Pentium 4 with 1GB of physical memory. For each benchmark, we report the average result of five runs; the observed variance was under 1%.

We compare runtime overhead against to the baseline GNU libc allocator, which is based on the Lea allocator [20] and is among the fastest general-purpose allocators [3].

Figure 11 shows that Plug degrades performance by 0% to 102% (`cfrac`), with a geometric mean of 17.7% across both benchmark suites. Plug's overhead is greater on the allocation-intensive benchmarks because its individual cost per allocation is higher than GNU libc. On SPECint, which is less allocation-intensive, Plug's runtime overhead averages only 3.0%.

6.2 Leak Tolerance

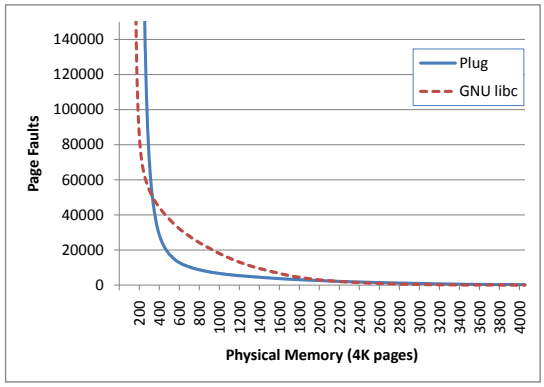
We quantify Plug's ability to tolerate memory leaks by measuring the increase in paging observed by the leaking application. Untolerated leaks increase the application's footprint until thrashing paralyzes the system and the application becomes unusable or crashes. Tolerated leaks can be written to swap and do not add to the footprint.

To measure paging performance, we use Pin [22] to track all memory references during execution. We feed the resulting trace into a simulator that generates histograms based on LRU position. Using these histograms, we generate *miss curves* that show the number of incurred page faults for any possible physical memory size.

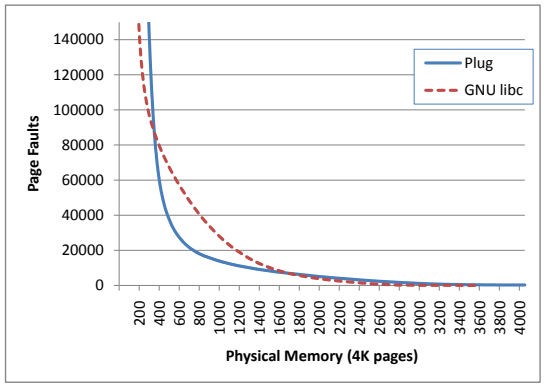
We evaluate leak tolerance using real-world leaks in a long-running server application taken from BugBench [21].

6.2.1 Squid

Squid is a web cache application which functions as an HTTP proxy. Client web browsers request pages from Squid, which it fetches from its in-memory cache or its local disk cache if available. On every request, Squid must consult the indices of these caches to see whether it can satisfy the request locally, or whether it must fetch the data from the hosting server.



(a) 50% leak rate



(b) 10% leak rate

Figure 12. Squid leak tolerance (see Section 6.2.1). Under moderate memory pressure, Plug allows Squid to run with far fewer page faults (up to 55% fewer).

Squid 2.3STABLE3 and earlier suffer a memory leak when handling SNMP requests. We test Plug's resilience against this leak by sending a sequence of 20,000 requests with varying mixtures of SNMP requests (leaks) and standard HTTP requests.

Figure 12 shows the number of page faults incurred by Squid under both the GNU libc allocator and Plug. We show results for request mixtures with 50% and 10% SNMP requests, where each such request leaks some memory. Under extreme memory pressure, Plug suffers more faults than GNU libc. We attribute this increase to the marginal increase in footprint caused by Plug's metadata. However, at these memory sizes, the fault rate would be so high that performance would be unacceptable under both allocators. At larger memory sizes, where no paging is occurring, libc is also competitive with Plug.

However, with moderate amounts of memory pressure, Plug's page fault rate is significantly lower than that of GNU libc (up to 55% fewer faults). This decrease is due to Plug's allocation site segregation: the leaked data never resides on pages containing cache index information. Once a page in the leaky site's heap is filled, it is never accessed again. Index data is kept dense in the default heap, keeping the working set size low despite the memory leak.

6.3 Virtual Compaction

Plug's mechanism for reducing physical memory consumption due to fragmentation is virtual compaction. One caveat of measuring the effectiveness of compaction is that the resident memory size reported by the kernel and shown in utilities such as `top` is incorrect when pages are mapped multiple times into virtual address space. The current kernel charges the process for each virtual mapping, rather than once per frame. However, the global count of physical memory reported is correct.

We verified this effect with a test program which repeatedly allocates objects and leaves pages with a single live object. This test case is the best case for virtual compaction. While the kernel reports the process as using gigabytes of physical memory, the reported global count (for all processes) is only a few hundred megabytes.

Since the kernel numbers are not accurate, Plug computes the memory savings of virtual compaction itself by tracking the difference between the number of virtual pages and actual physical pages allocated for its heap. The resident set size we report is the RSS reported by the kernel minus the amount of savings Plug calculates.

Firefox Memory Overhead

To measure the effectiveness of virtual compaction on large programs, we compare the resident memory size of Firefox running under the GNU libc allocator to Plug, configured both with and without virtual compaction enabled.

For each experiment, we loaded the same series of 25 pages into several tabs in a single browsing session. We then measured the memory consumption reported by `top`.

	Virtual Address Space	Physical Memory
GNU libc	218M	86M
Plug	233M	107M

Figure 13. Firefox Memory Consumption

Figure 13 shows the result of the experiment. Plug requires 24% more physical memory than GNU libc. Virtual compaction saves over 2100 pages (8.5 MB), about 7% of the total physical memory requirement. However, only 49M of memory is used for age-segregated heaps, so virtual compaction achieves an almost 20% reduction in the amount of physical memory consumed by these heaps.

Virtual Compaction in Small Programs

The bulk of the allocation-intensive benchmarks primarily allocate short-lived objects, so few age-segregated heaps are created, and the lifetimes of these objects tend to be short. However, virtual compaction has a significant effect on the memory usage of `cfrac`. While `cfrac` is short-running (around 5 seconds), virtual compaction reduces the total number of physical pages by 47% (from 2726 pages to 1425 pages).

7. Related Work

We describe related work first by drawing the distinction between two key types of memory leaks, and then discuss previous work and how it addresses these leaks.

Categories of leaks

Memory leaks fall into two classes: *reachability* leaks and *staleness* leaks. A reachability leak is the classic leak scenario in C/C++ applications: the program loses the last pointer to an object without

calling `free` on it. Staleness leaks occur when the program inadvertently holds a pointer to an object that it will never again access. These leaks can be problematic even in language with garbage collection, since they are still reachable from live data in the heap.

Leak tolerance

To the best of our knowledge, the only previous work on tolerating leaks in unmanaged languages is Cyclic Memory Allocation (CMA) [29]. CMA avoids leaks by replacing dynamically-allocated memory with fixed-size buffers based on profiling runs. CMA can only eliminate leaks from sites which it identifies as bounded and can erroneously overwrite live data when profiling is incorrect.

Melt tolerates staleness leaks in Java [8]. Melt's approach is similar in flavor to Plug, in that it segregates live data from leaked objects, compacts leaked objects onto the same page, and allows those pages to be swapped to disk. However, Melt targets Java applications, and so can use a moving collector to perform object segregation on demand. Because C and C++ do not allow object relocation, Plug instead segregates objects at allocation time and introduces virtual compaction to allow objects to be compacted without being moved.

Garbage collection (GC) tolerates reachability leaks, which it automatically reclaims, but does not address staleness leaks. Though garbage-collected languages like Java and C# have seen widespread use, the adoption of garbage collection for C and C++ applications has been limited. Despite its benefits, practical collectors for these languages have both real and perceived drawbacks, both for performance and correctness. Pointer misidentification can cause conservative garbage collectors to fail to reclaim memory, especially on 32-bit platforms [5]. Worse, because C/C++ programs can obscure pointers (e.g., via XOR-encoding of linked lists [37]), a conservative collector can inadvertently reclaim live objects, causing these programs to crash.

Static analysis can also eliminate memory leaks by program transformation. Shaham et al. present two analyses which can eliminate memory leaks in Java: the first detects dead entries in arrays that will never be read in the future [33], while the second uses shape analysis to detect dead references [34]. Lattner and Adve propose pool allocation, a transformation that can statically eliminate some leaks in C/C++ applications via points-to set liveness [19].

Dynamic leak detection

Most research on memory leaks has focused on the problem of detection rather than tolerance. Previous work on detection for C and C++ applications falls into two categories. First are tools that find *reachability leaks*, which are objects that the application no longer holds a pointer to, but are unfreed. Conservative garbage collection techniques can be used to find unreachable objects. Several tools use this approach, including Purify [11], Valgrind [28], and RADAR [25]. While these tools are useful for diagnosing a large class of leaks, they cannot find leaked objects that are still reachable.

The second category of leak diagnosis tools are based on measuring object *staleness*. These tools find both reachability and staleness leaks, as they detect actual use of object by the program. One drawback of these tools is *incompleteness*, that is, they may produce false positives. This drawback is a byproduct of the approach: in the general case, no tool can determine with certainty whether or not the program will use a given object in the future. However, finding stale objects has proven to be a useful approach for diagnosing leaks.

Hauswirth and Chilimbi's SWAT leak detector estimates object staleness using program instrumentation [12]. Precisely determining staleness requires tracking every read and write to the heap, which causes unacceptable overhead. To reduce overhead to acceptable levels, SWAT uses *code sampling*. Instrumentation is enabled randomly, biased towards infrequently executed code. Bond and McKinley propose Sleigh [7], a leak detector for Java roughly similar to SWAT. Qin et al. propose SafeMem, another leak detector based on object staleness [31]. Rather than code instrumentation, SafeMem uses ECC memory in a novel way to detect memory accesses.

Several papers focus on providing more detailed information about the causes of leaks in an effort to reduce the burden of fixing them. Mitchell and Sevitsky's LeakBot automatically identifies Java data structures that are likely to be the cause of leaks by evaluating the evolving structure of the heap graph [26]. Jump and McKinley describe a low-overhead approach to inferring sources of leaks by examining dynamic characteristics of the points-from graph in Java programs [17]. Maebe et al. describe a high-overhead leak detector that identifies the specific program statement responsible for removing the last reference for reachability leaks [24].

Static leak detection

Static analysis can detect certain types of memory leaks, but suffer from false positives due to analysis imprecision. Clouseau infers ownership constraints and finds violations which may indicate leaks [13]. Xie and Aiken use boolean constraints to find leaks based on escape analysis [39]. Cherem et al. propose an analysis that considers flows through the program graph from allocation points to deallocation points to identify possible leaks [9]. Orlovich and Rugina's analysis proves the absence of leaks, but can be used to detect leaks when the proof fails [30].

VM-techniques for memory management

Dhurjati and Adve introduced a technique for detecting dangling pointer errors that also uses virtual memory remapping primitives [10]. In their system, every object is allocated on a new virtual page, with multiple virtual pages mapped to the same physical page to conserve space. By protecting the virtual pages holding individual objects whenever they are freed, their system can detect all dangling references. By contrast, virtual compaction starts with many objects mapped to individual virtual pages, and later combines the virtual pages (holding multiple objects) onto a single physical page.

Recent cooperative systems exploit communication between the OS virtual memory manager (VMM) and the garbage collector to reduce paging due to garbage collection. Yang et al. modify the Linux virtual memory manager to provide detailed reference information, allowing it to dynamically adapt the GC heap size in order to maximize performance [40]. Plug uses a derivative of CRAMM's mechanism to control the size of its aging queues. Hertz et al. present the bookmarking collector, a cooperative system where the OS informs the runtime system of impending page eviction, and the garbage collector summarizes information on the pages ("bookmarks") that allow it to avoid traversing paged-out memory during garbage collection. Archipelago [23] uses an object-per-page allocator to improve resilience against buffer overflow errors and uses virtual memory protection to compact cold pages and reduce physical memory overhead.

Appel and Li describe a number of primitives and algorithms for exploiting virtual memory in user-mode [1]. Plug follows in the spirit of those algorithms, and makes use of many of the described primitives.

8. Future Work

While Plug tolerates memory leaks by preventing performance degradation, it does not prevent against exhaustion of virtual address space, which causes the program to crash. Currently, Plug

is *sound*, in that its toleration mechanisms will never cause a correct program to fail. Unsound techniques such as cyclic memory allocation [29] prevent virtual address space exhaustion at the expense of correctness. We plan to extend Plug with an unsound option to unsoundly free memory when a leak causes the program to exhaust its virtual address space. Plug already has a page-protection mechanism used to estimate page staleness which we can exploit to choose the best data to unsoundly free.

Plug has a simple mechanism to prevent virtual compaction from merging hot and cold data. However, if program behavior changes (such as entering a new program phase), cold data may become hot again. We plan to make Plug's heuristics for merging more robust to this phenomenon as well as enable Plug to split apart pages which become mixed, thus re-segregating hot and cold data.

Our current prototype relies on the operating system to evict leaked data to swap, freeing physical memory. Plug could use compression to reduce swap consumption. Many leaked pages will contain few objects or substantially similar objects that may compress well.

9. Conclusion

This paper presents Plug, a memory allocator and runtime system that tolerates memory leaks by reducing performance degradation due to paging by up to 55%. Plug's key contribution is its hybrid memory management scheme, which both segregates objects at allocation time with a *context-sensitive* allocator and separates leaks from non-leaked objects with an *age-segregated* allocator. A novel *virtual compaction* mechanism allows Plug to compact memory without the need to move objects, reducing the fragmentation due by segregation without degrading Plug's ability to tolerate leaks.

Plug operates on unaltered binaries, making deployment simple. Unlike garbage collection, Plug tolerates both reachable and unreachable leaks. For a range of applications, including servers and applications with low allocation-intensity, Plug incurs minimal runtime and memory overhead, making it practical for use even for large deployed applications where performance is a key concern.

10. Acknowledgments

The authors would like to thank Ting Yang and Scott Kaplan for valuable discussions about Plug and Linux virtual memory management, Ted Hart for his feedback during the development of Plug, and Shan Lu, Martin Rinard, and Huu Hai Nguyen for providing us the leaky inputs for *squid*.

References

[1] A. W. Appel and K. Li. Virtual memory primitives for user programs. *ACM SIGPLAN Notices*, 26(4):96–107, 1991.

[2] E. D. Berger and B. G. Zorn. Efficient probabilistic memory safety. Technical Report UMCS TR-2007-17, Department of Computer Science, University of Massachusetts Amherst, Mar. 2007.

[3] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 2002*, Seattle, Washington, Nov. 2002.

[4] 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.

[5] H. J. Boehm. Space efficient conservative garbage collection. *SIGPLAN Not.*, 39(4):490–501, 2004.

[6] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[7] M. D. Bond and K. S. McKinley. Bell: bit-encoding online memory leak detection. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.

[8] M. D. Bond and K. S. McKinley. Tolerating memory leaks. Technical Report TR-07-64, Department of Computer Sciences, University of Texas at Austin, Dec. 2007. Available at http://www.cs.utexas.edu/~mikebond/melt-tr-2007.pdf.

[9] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 480–491, 2007.

[10] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 269–280, Washington, DC, USA, 2006. IEEE Computer Society.

[11] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, California, 1991.

[12] 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 Press.

[13] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 168–181, 2003.

[14] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, San Diego, CA, Oct. 2005.

[15] M. Hirzel and A. Diwan. On the type accuracy of garbage collection. In *ISMM '00: Proceedings of the 2nd International Symposium on Memory Management*, pages 1–11, New York, NY, USA, 2000. ACM.

[16] R. E. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996.

[17] M. Jump and K. S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–38, 2007.

[18] P.-H. Kamp. Malloc(3) revisited. http://phk.freebsd.dk/pubs/malloc.pdf.

[19] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, 2005.

[20] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[21] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[23] 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.

[24] J. Maebe, M. Ronsse, and K. D. Bosschere. Precise detection of memory leaks. In *Workshop on Dynamic Analysis (WODA04)*, pages 25–31, 2004.

[25] Microsoft TechNet, Microsoft Corporation. *Memory Leak Diagnoser*, Dec. 2007.

[26] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[27] Mozilla.org. Bugzilla@mozilla, 2008. [Online; accessed 12-March-2008].

[28] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM Press.

[29] 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, 2007.

[30] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *SAS '06: Proceedings of the 13th Annual Static Analysis Symposium*, pages 405–424, 2006.

[31] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. *HPCA*, 00:291–302, 2005.

[32] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[33] R. Shaham, E. K. Kolodner, and S. Sagiv. Automatic removal of array memory leaks in java. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 50–66, London, UK, 2000. Springer-Verlag.

[34] R. Shaham, E. Yahav, E. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *SAS '03: Proceedings of the 10th Annual Static Analysis Symposium*, 2003.

[35] Standard Performance Evaluation Corporation. SPEC2000. http://www.spec.org.

[36] L. Torvalds. Linux kernel mailing list post. http://lkml.org/lkml/2004/1/12/265.

[37] Wikipedia. XOR linked list — wikipedia, the free encyclopedia, 2008. [Online; accessed 19-March-2008].

[38] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116, Kinross, Scotland, Sept. 1995. Springer-Verlag.

[39] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 115–125, 2005.

[40] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2006.

[41] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 2004 ACM SIGPLAN International Symposium on Memory Management (ISMM 2004)*, pages 61–72. ACM Press, Nov. 2004.