Efficiently and Precisely Locating Memory Leaks and Bloat

Gene Novark Emery D. Berger
Department 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

Inefficient use of memory, including leaks and bloat, remain a significant challenge for C and C++ developers. Applications with these problems become slower over time as their working set grows and can become unresponsive. At the same time, memory leaks and bloat remain notoriously difficult to debug, and comprise a large number of reported bugs in mature applications. Previous tools for diagnosing memory inefficiencies—based on garbage collection, binary rewriting, or code sampling—impose high overheads (up to 100X) or generate many false alarms.

This paper presents Hound, a runtime system that helps track down the sources of memory leaks and bloat in C and C++ applications. Hound employs *data sampling*, a staleness-tracking approach based on a novel heap organization, to make it both precise and efficient. Hound has no false positives, and its runtime and space overhead are low enough that it can be used in deployed applications. We demonstrate Hound's efficacy across a suite of synthetic benchmarks and real applications.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Reliability; D.2.5 [*Software Engineering*]: Debugging aids; D.3.3 [*Programming Languages*]: Dynamic storage management

General Terms Algorithms, Languages, Reliability

Keywords Hound, virtual compaction, dynamic memory allocation, memory leak detection, heap profiling

1. Introduction

Memory management is a notorious source of problems in C and C++. Even when programmers manage memory *correctly*—avoiding potentially catastrophic errors like double or invalid frees and dangling pointer errors—it remains challenging for them to use this memory *efficiently*. Memory inefficiency occurs whenever a program consumes more memory than it actually needs. When a program has unnecessary excess memory consumption, the program exhibits *bloat*. If a program has unneeded memory—memory that it will never use again—and never reclaims it, the program has a *leak*.

Inefficient use of memory reduces both performance and availability. Bloated applications limit the number of applications a user can run or degrade responsiveness by forcing other applications to be paged out. Memory leaks cause an application's memory image to grow over time, eventually triggering paging. Leaky applications thus become slower and slower, eventually becoming unresponsive or exhausting available address or swap space.

Perhaps because of their impact on usability, memory inefficiencies continue to be one of the most common classes of reported bugs. Both memory leaks and bloat are notoriously difficult to detect and debug in long-running applications, such as servers or web browsers. For example, in the first two months of 2008, over 150 memory leak bugs were reported in the Firefox browser [20].

Debugging memory inefficiencies is difficult for several reasons. First, some existing tools do not provide enough information or cannot be used in deployment. Existing *tracing-based* tools—like Purify [7] and Valgrind [21]—cannot locate reachable leaks or identify bloat. Worse, their high overheads (up to 100X) prevents their use in deployed applications. Because leaks often manifest over long periods of time and in response to exceptional events, it can be extremely difficult for developers to reproduce them in-house.

Second, while newer tools both can be used in deployment and detect all forms of memory inefficiency, they can produce too many false alarms. These *staleness*-based tools track the amount of elapsed time since the last access to each object [3, 8], allowing them to identify both bloat and leaks,whether reachable or not. However, because exact staleness tracking would be prohibitively expensive (requiring instrumentation of every memory access), these tools use adaptive *code-based* sampling techniques: the more frequently a code path is executed, the lower the chance it will update access information. However, code sampling can overestimate object staleness and thus generate many false alarms. Because programmer intervention is required to separate real errors from false positives, a high false alarm rate can make these tools unusable.

Contributions

This paper presents **Hound**, a runtime system for C/C++ that locates both leaks and bloat *without any false alarms*. Hound uses *data sampling* instead of code sampling to track object staleness with high precision while maintaining sufficiently low runtime and memory overhead for production use. To accomplish this, Hound relies on a combination of virtual memory primitives and novel memory management techniques: **context-sensitive** memory allocation and **age-segregated** memory allocation, which enable tracking and isolation of stale objects, and **virtual compaction**, which reduces Hound's memory consumption by compacting objects in *physical* but not *virtual* memory, making it suitable for C and C++.

We demonstrate Hound's low overhead across a suite of standard benchmarks and applications with real memory leaks. For non-allocation-intensive workloads, Hound imposes low runtime

PLDI'09, June 15–20, 2009, Dublin, Ireland.

overheads, averaging less than 5% across a suite of server applications, all of SPECint2000, and all but two of the benchmarks from SPECint2006. We show that, despite this low overhead, Hound generates precise leak reports (i.e., with no false positives) that can point directly to the source of memory inefficiencies in both synthetic and real applications.

Outline

The rest of this paper is organized as follows. Section 2 first provides an overview of Hound's key algorithms. Section 3 contrasts code sampling, used in previous tools, with Hound's data sampling. Section 4 describes Hound's segregating allocator in detail, and Section 5 describes how Hound tracks object staleness. Section 6 describes the virtual compaction mechanism Hound uses to reduce memory overhead. Section 7 describes how Hound generates its reports. Section 8 empirically evaluates Hound's overhead and its precision at locating memory inefficiencies. Section 9 presents an overview of related work, and Section 10 concludes.

2. Overview

Hound relies on memory protection to detect object accesses and estimate staleness. This mechanism works at a page granularity, meaning Hound either enables or disables tracking for all objects on a page as a group. In order to accurately estimate staleness with low overhead, Hound collocates objects with similar predicted behavior on the same page. This segregation allows constant protection and monitoring of rarely-used objects, while hot objects are grouped together on the same pages and can be left unprotected.

Hound's allocator segregates objects along two different axes. First, Hound uses a **context-sensitive** allocation strategy: it uses the calling context of `malloc` calls—known to be an effective predictor of object lifetime [27]—to segregate objects from different *allocation sites* onto different pages. This policy groups similarly behaved objects onto a small number of pages and prevents dissimilar objects from being allocated into those pages. While the effectiveness of this policy depends on program behavior, our results show the approach is effective in real programs.

Hound combines this context-sensitive allocation with an **age-segregated** memory allocator. Age segregation ensures that all objects on the same page are of similar age, as measured by allocation time. Eventually, as the program frees non-leaked objects, leaked objects and bloat will be isolated on their own pages.

Hound further refines these algorithms to reduce the risk of excessive fragmentation. First, Hound 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 each call site with only a single live object holding down an entire page. Notice that this policy does not impair Hound's precision: by definition, sites that allocate few objects cannot be the source of significant memory leaks.

Second, Hound performs **virtual compaction**, a novel technique that compacts memory without the need to move objects (something C and C++ do not permit). Virtual compaction retains segregation within the *virtual* address space, while significantly reducing the *physical* fragmentation of the heap by merging sparsely-populated virtual pages onto the same physical page frame. Thus, virtual compaction can reclaim a substantial amount of physical memory when live objects are sparsely spread across many pages. This technique reduces the amount of physical memory required for age-segregation without sacrificing the precision of Hound's object tracking.

3. Code vs. Data Sampling

In contrast to leak detection tools that use *code-based* sampling, Hound uses a *data-based* sampling technique. Data sampling prevents overestimating object staleness, a source of false positives in tools based on code sampling. This section explains the problems inherent in using code sampling for leak detection and presents Hound's solution.

State-of-the-art leak detection tools such as SWAT [8] and Sleigh [3] use a code sampling technique called *adaptive bursty tracing* (ABT). This technique samples code inversely proportional to its execution frequency. Thus, infrequently-executed code is traced nearly all the time, while frequently-executed code is rarely sampled. This strategy allows thorough tracing of code where bugs may lie, while avoiding the overhead of tracing well-tested hot code paths.

Using ABT to sample object accesses for staleness estimation relies on the assumption that hot code paths access hot objects, and that cold code paths access cold objects. Accesses to objects during hot code execution will be frequently missed due to sampling. If a program accesses the same object frequently, then sampling will detect some references to it, correctly estimating staleness. However, if a hot code path accesses many different objects, most of the objects will have greatly overestimated staleness values.

For example, consider a large hash table accessed only through a single hot code path. If ABT uses a minimum 0.1% sampling rate (as in the experiments in the SWAT paper), then it will miss all but 1 reference out of every 1,000. Assuming a random distribution of accesses, each individual object in the hash table is accessed only a small fraction of the total accesses to the table. As a result, most objects in the table appear far staler (1,000X) than they actually are, even though they are live and referenced relatively often.

Instead of code sampling, Hound uses a *data sampling* approach which avoids overestimating staleness. Hound periodically protects every page and updates the last access time of all objects on that page to the current allocation time. When a protected page is accessed, Hound unprotects it and places it on an active list. Protected pages cannot have been accessed since their most-recent protection time, providing a strict lower bound on the staleness of all objects on the page.

Hound's sampling policy cannot result in false positives. However, it may result in false negatives when a hot object is collocated on a page with stale objects. Hound uses an aggressive segregation policy to reduce the frequency of these situations, as the next section describes.

4. Hound Heap Structure

Hound relies on object segregation to achieve low-overhead staleness detection using memory protection. Without segregation, stale objects would often be on the same page as frequently-accessed objects. Thus, Hound could not protect stale objects without protecting hot objects, and would thus incur a high page-fault rate, causing unacceptable performance degradation.

While garbage-collected languages like Java support moving garbage collection [10] and can thus segregate objects at GC time, C's and C++'s direct access to memory addresses precludes object relocation. Thus, the only way for Hound to segregate objects is to separate them at allocation time.

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

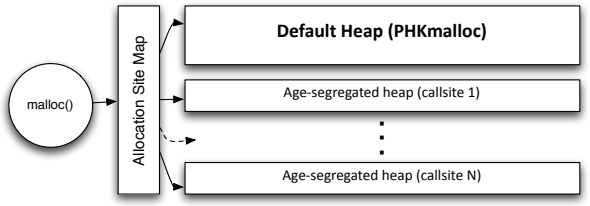

Figure 1. Hound's context-sensitive heap structure, which segregates allocation requests by *allocation site* once the number of live objects exceeds a fixed threshold (see Section 4.1).

```
1  void * houndmalloc (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->getAgeHeap() != NULL) {
10     return m->getAgeHeap()->malloc (size);
11   } else if (m->getLiveCount() >= 64) {
12     // make a new heap.
13     m->initAgeHeap();
14     return m->getAgeHeap()->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 Hound's allocation-site segregated `malloc`.

```
1  void houndfree (void * ptr) {
2    // check pointer validity.
3    if (!isFromHoundHeap(ptr)) {
4      // return to standard allocator
5      // after updating metadata.
6      int context = getHeader(ptr);
7      Metadata * m = getMetadata(context);
8      m->liveCount--;
9      phkfree_with_header (ptr);
10   } else {
11     // locate the page via ptr masking
12     // and free the object.
13     void * page = ptr & ~(PAGE_SIZE-1);
14     PageEntry * entry = pageMap(page);
15     entry->free (ptr);
16   }
17 }
```

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

4.1 Allocation-Site Segregation

Previous research has shown that objects allocated from the same call site tend to exhibit similar behavior and lifetime patterns [27]. To isolate leaks and bloat, Hound segregates objects by associating a separate heap with each allocation site. Hound identifies these sites with bounded context sensitivity (the last four return addresses on the call stack).

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

The result is that pages tend to fall into two classes: those that contain all cold objects, which can be constantly protected without incurring page faults, or mostly hot objects, which are left unprotected, and may increase 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 a typical interaction. However, most of those sites create relatively few objects, especially those sites that correspond to its initialization phase. Allocating an entire page to hold a few small objects wastes memory.

To reduce this memory overhead, Hound instantiates a new heap for a site only when the number of live objects from that site reaches some threshold (currently 64). To track the count for each site, Hound adds an extra header word to each object containing the hash of the allocation site of each object. When an object is freed, Hound decrements the live count for its allocation site.

At allocation time, Hound 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, Hound allocates object requests directly from a conventional heap (line 18); Hound currently uses PHKmalloc [12] as its allocation substrate. 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 or bloat.

To further reduce memory overhead, we modified PHKmalloc slightly. PHKmalloc normally uses `sbrk` to allocate memory, but this approach can prevent reclamation of freed objects if subsequently-allocated objects remain live. We changed PHKmalloc to allocate large objects (at least 64K) directly from the system via `mmap`, allowing their space to be returned to the system as soon as they are freed. This hybrid allocation strategy is used by many memory allocators, including DLmalloc and Hoard.

4.2 Age-Based Segregation

While call site segregation is a heuristic that can help separate objects with similar behavior, some sites may still be heterogeneous. For example, a site may generate objects that eventually become stale as well as short-lived objects. Allocating a short-lived object onto a page with stale objects would cause Hound to underestimate the staleness of the other objects on that page.

To avoid this scenario, Hound departs further from conventional heap layouts by segregating objects by *age*, as measured in allocation time. The key insight is the following: objects that are leaks or bloat by definition are not reclaimed for a long time, and thus become older and older as program execution continues. Keeping old objects separate from newer objects thus prevents stale objects from intermingling with newer, potentially hot objects.

While age segregation can ensure accurate staleness information in the case of some anomalous sites, other pathologies may lead to incorrect staleness information. For example, if a site produces many stale objects and some long-lived hot objects, the hot

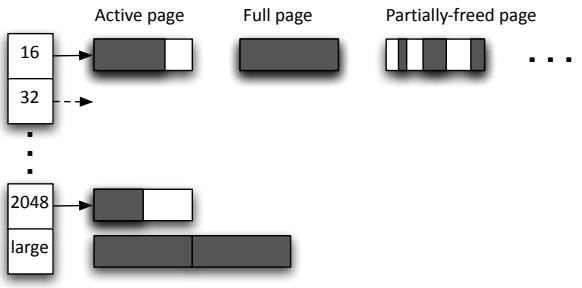

Figure 4. Hound's age-segregated heap (see Section 4.2).

objects will cause Hound to incorrectly record that many of the stale objects are active due to page-level false sharing. Note that this inaccuracy only *underestimates* staleness, and thus cannot lead to false alarms.

Hound separates young from old objects in an **age-segregated heap**. Each age-segregated heap is itself a segregated-fits allocator [35] 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.

Hound 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, Hound 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). Hound 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). Hound adds filled pages to the aging queue, where they will eventually be protected for staleness tracking, and also become candidates for virtual compaction, described in Section 6 (lines 9–12).

Hound 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 Hound free pages when their population drops to zero; and (3) a bitmap that tracks which slots contain live objects, used by Hound's virtual compaction algorithm. Hound uses a two-level page table structure to map page addresses to metadata.

5. Tracking Staleness

Hound keeps all filled pages (those that will not be used for subsequent allocations) on its *aging queue*. This aging queue is organized in order of staleness, measured as the time since the application last accessed some object on the page. Hound protects pages on the queue from direct read and write access using the `mprotect` system call. Objects on the page cannot have been accessed since the last time the page was protected. Hound estimates staleness using the formula `currentTime` − `protectionTime`, which provides a strict lower bound on true staleness. When the application accesses an object on the page, Hound handles the page fault (SIGSEGV) and unprotects the pages. While unprotected, all objects on the page are considered to have a staleness value of 0.

Protecting all pages on the aging queue would be prohibitively expensive, since some pages will contain frequently-used objects. Hound 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 a FIFO queue. When the program accesses a page on the inactive list, a page fault occurs and Hound unprotects and

```
1  void * AgeHeap::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 Hound's age-segregated `malloc`.

```
1  void PageEntry::free (void * ptr) {
2    // find originating page.
3    inUse--;
4    bitmap.clear(indexOf(ptr));
5    if ((inUse == 0) && (bump == endOfPage)) {
6      // free the page for re-use.
7      recyclePage (page);
8      clearPageEntry (page);
9    } else if ((inUse < NUM_ENTRIES/2)
10         && (bump == endOfPage)) {
11     // check virtual compaction queue
12     AgingQueue.addOrUpdate(this);
13   }
14 }
```

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

moves the page to the head of the active list. Hound periodically moves pages from the end of the active list onto the inactive list to maintain the latter's target size.

5.1 Adapting the Inactive List

The size of the inactive list is controlled adaptively to achieve both acceptable runtime overhead and to maximize the useful information acquired. 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. Hound's heuristics for controlling the list sizes and moving objects from the active to inactive list are based on those used in CRAMM [37] and shown as pseudocode in Figure 7.

Each time a page is added to the aging queue, Hound checks whether it should adjust the sizes of the queues. If 1/8 second of CPU time has passed (or 10 page faults), Hound reevaluates the

```
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 7. Pseudo-code for Hound's segregated aging queue.

sizes. Hound 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, Hound decreases the target size of the inactive list. If it is less than 0.5%, Hound increases its size.

Hound maintains a *target inactive size*, initially 0. When changing the size of the inactive list, Hound 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.

Hound'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. Hound 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. Choosing the smaller of P_I and P_A ensures that small changes are made when either list is small, preventing sudden, drastic changes in behavior.

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, Hound checks the current inactive list size against the target. If the current list is too small, Hound moves up to 8 pages from the active list to the inactive list. When the target size is smaller than the current size, Hound 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 have accurate staleness information.

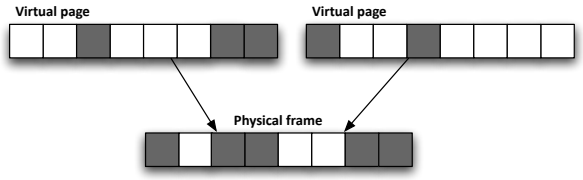

Figure 8. An example pair of pages that share no common live indices. The virtual compactor can merge these pages in physical memory while not actually relocating objects in virtual memory (see Section 6).

6. Virtual Compaction

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

To mitigate this problem, Hound uses a novel scheme we call **virtual compaction** that leverages Linux's virtual memory remapping capability 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 Hound, 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 Hound's age-segregated heaps, which use a segregated fits structure in which each page is an array of identically-sized objects. For each page, Hound maintains a bitmap indicating which positions within the page are occupied by live objects. If a pair of pages contain live objects only at different positions (i.e., there is no offset containing 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 8). 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, Hound 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.

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

6.1 Finding Candidate Pairs

At runtime, Hound'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. Hound therefore makes no attempt to optimally compact pages, and instead relies on heuristics that are effective in practice (see Section 8.3).

```
// 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)) {
      // virtually compact p with q.
      p.mergeWith(q);
      return;
    }
  }
}
```

Figure 9. Pseudo-code for Hound's first-fit virtual compaction algorithm (see Section 6).

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

Hound considers compaction only for pages which have less than 50% occupancy. It tracks these pages using its **fragmentation manager** which keeps a linked list of low-occupancy pages per size class. 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.

6.2 Merging Pages

When merging pages, Hound 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 [32]. 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 do not conflict.[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.

7. Reporting

While Hound's staleness tracking operates on the page level, it produces reports that summarize staleness information per allocation site. These reports present each allocation site, ranked by the severity of their memory consumption, and provide information the programmer can use to diagnose leaks or other inefficient usage of memory.

Figure 10 shows two allocation sites from Hound's report for the Squid web cache. The report produces 7 total sites, but the remaining 5 have few live objects. The first reported site is a true memory leak, while the second is caused by data structures which are eagerly allocated at the start of the program and never used, and may thus be unneeded bloat. Hound ranks allocation sites by their total *drag*, the sum of object size (in bytes) times staleness (in

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

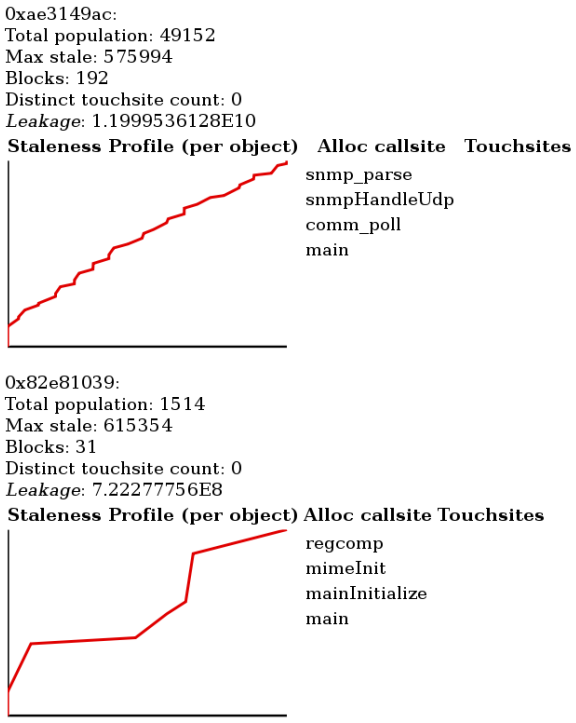

Figure 10. Hound's report for Squid

allocation time) for each unreclaimed object from that site [26]. It also reports the total number of objects and blocks and the maximum staleness of any object from the site. It also shows the calling context of the site as well as the last-reported touch sites of the pages (the calling context of the instruction which caused a protection fault on the page).

The graph presented with each allocation site presents a cumulative distribution function (CDF) of the staleness of the pages from the site. These graphs provide a visual representation of how the staleness behavior of objects from the callsite. A point on the curve at (x, y), considering the axes ranging from 0–100%, means that $y\%$ of pages have staleness $x\%$ of the maximum for that site. Intuitively, a line close to diagonal, such as the first site in Figure 10 (a real leak), represents a leak that increases steadily over time. The second site shows a large fraction of objects (around a third) that are not stale (the plateau in the lower left), and then that most of the rest of the objects are quite stale (the sudden jump in the center).

Hound produces detailed information on every block in heap. The reporting tool uses these raw dumps to generate the summary information. However, the programmer may find the raw dumps to be useful, as they present exact information on the number of objects surviving on every page and the individual staleness estimates. The raw information is produced as an XML file, making it easy to use existing tools to mine the data. The tool that produces our visual reports is written in the XQuery language and transforms the raw XML into an XHTML/SVG report.

8. Experimental Results

Our evaluation answers the following questions:

1. What is Hound's runtime overhead?
2. What is Hound's memory space overhead, and how effectively does virtual compaction reduce its physical memory consumption?

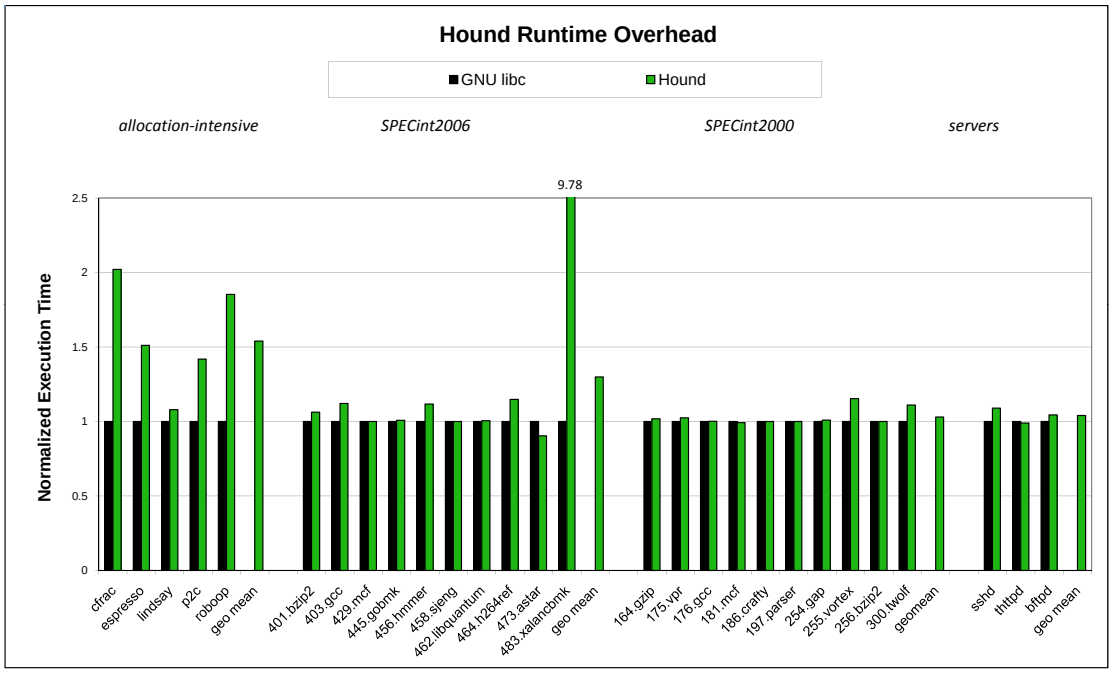

Figure 11. Runtime overhead for Hound across a suite of benchmarks, normalized to the performance of the GNU libc (Linux) allocator (see Section 8.1). Hound incurs minimal overhead for the SPECint2000 and SPECint2006 benchmark suites of CPU-intensive applications and for a range of server applications, though its overhead is substantial for allocation-intensive benchmarks.

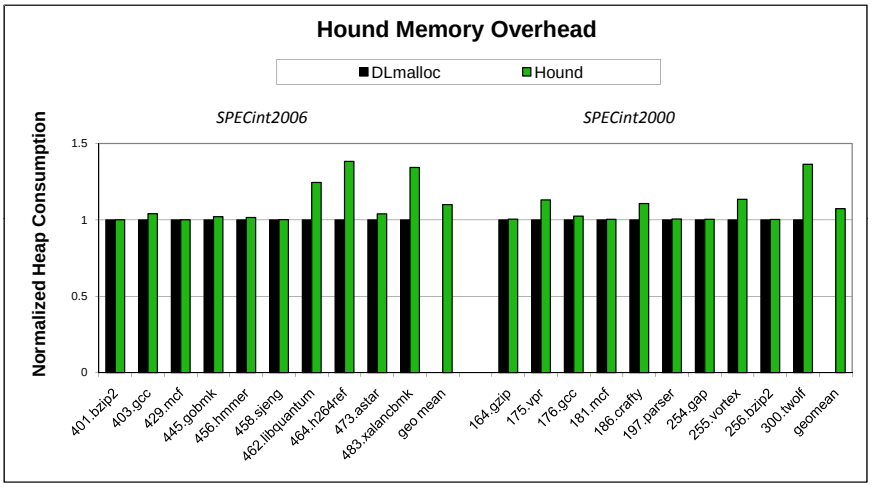

Figure 12. Memory overhead for Hound across the SPECint2000 and SPECint2006 suite of benchmarks, normalized to the consumption of the DLmalloc allocator (see Section 8.2). Hound incurs minimal memory overhead for most of the benchmarks.

3. How precisely does Hound compute object staleness while avoiding false positives?
4. How well does Hound identify callsites corresponding to leaks, and what is its false positive rate?

8.1 Runtime Overhead

We evaluate Hound's runtime performance on several benchmark suites. The first is a range of highly allocation-intensive benchmarks. These benchmarks allocate and deallocate objects at unusually high rates, and as such stress Hound's allocation mechanisms. While these applications are not generally representative of typical workloads due to their high allocation rates, we include them here because they have been widely used in previous memory management studies.

The second set of benchmark suites is the SPECint2006 and SPECint2000 suites of CPU-intensive benchmarks [30], which we run using their reference workloads.[2].

We also evaluate Hound's performance with three different server applications: the *thttpd* web server, the *bftpd* ftp server, and the *OpenSSH* server. For the first two, we record total throughput

[2] `471.omnetpp` from SPECint2006 times out, `252.eon` from SPECint2000 fails on both GNU libc and Hound, and the perl benchmarks from both suites fail to run with our current implementation of Hound.

achieved with 50 simultaneous clients issuing 100 requests each. For OpenSSH, we record the time it takes to perform authentication, spawn a shell, and disconnect.

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 DLmalloc [14] and is among the fastest general-purpose allocators [2].

Figure 11 shows Hound's overhead across the benchmark suites. Hound's overhead is substantial on the allocation-intensive benchmarks (from 7.9% to 102%, with a geometric mean of 54%), because the cost of each allocation and deallocation is higher than in GNU libc.

However, Hound's runtime overhead is generally far lower for the other benchmark suites. On SPECint2006, except for `omnetpp` and `xalancbmk`, Hound's overhead ranges from -9% to 15%, with a geometric mean of 4%. The allocation patterns of `omnetpp` and `xalancbmk` heavily stress Hound's virtual compaction mechanism by having many sparsely-populated pages during most of the benchmark run. The simple matching algorithm used in the current implementation results in significant overhead while scanning the list of pages. More robust algorithms for finding candidate pairs is an area for future work. The `omnetpp` benchmark has a timeout mechanism that causes premature termination due to the overhead imposed by Hound.

On SPECint2000, Hound's overhead ranges from 0% to 15%, with a geometric mean of 3%. On the server benchmarks, the overhead ranges from -1% to 8%, with a geometric mean overhead of 4%. We believe that these ranges are likely to be typical of long-running applications, which allocate memory at a far lower rate than the allocation-intensive benchmarks.

8.2 Memory Overhead

We evaluate Hound's memory overhead compared to DLmalloc [14], the basis for the GNU libc allocator. We measure total virtual memory consumption of the heap using a tool that observes all calls to `mmap` and `sbrk` and accounts for Hound's use of `mremap`. It reports the maximum virtual memory consumption (high water mark). We use the DLmalloc itself rather than GNU libc in this experiment because libc appears to call internal versions of `sbrk` which cannot be shimmed by standard methods (e.g. `LD_PRELOAD`).

Figure 12 shows relative memory consumption for SPECint2006. The other benchmark suites (allocation-intensive and server) consist of applications with very small heap footprints, at most 2MB under Hound. Hound requires several hundred kB of memory for global metadata regardless of actual application memory usage, making relative comparison misleading for small programs. The SPEC benchmarks require much more heap memory (mean: 250MB), though most have lower allocation rates in terms of `malloc` calls per second.

Because some benchmarks consist of multiple executions on different inputs, we measure the overhead required for the largest input. For most benchmarks, Hound imposes minimal memory overhead: 7 of 10 SPECint2006 benchmarks need less than 5% more heap space when running under Hound than with DLmalloc. Of the remainder, `xalancbmk` requires 34% more, `libquantum` 25%, and `h264ref` 13%. SPECint2000 shows similar overheads, with most benchmarks requiring less than 15% more memory. The sole exception is `twolf`, which has a small heap footprint of only 3.4 MB under DLmalloc. Under Hound, it requires 4.6 MB, an increase of 36%.

8.3 Virtual Compaction

We evaluate Hound's ability to limit fragmentation via its virtual compaction mechanism on the Firefox browser as well as on the small, allocation-intensive benchmark, `cfrac`.

Firefox Memory Overhead

To measure the effectiveness of virtual compaction on large programs, we compare the memory requirements of Firefox (version 2.0.0.9) running under DLmalloc to Hound, configured both with and without virtual compaction enabled. We measure both the total virtual memory consumption of the entire process, and the physical memory required by the heap alone.

For each experiment, we loaded the same series of 25 pages during a single browsing session. We then measured virtual memory consumption using `top` and heap usage using the tool described in Section 8.2.

Under DLmalloc, Firefox's heap requires 239 MB of memory. Under Hound, it rises to 267 MB, an increase of 11%. Virtual compaction saves over 4,000 pages (16 MB), about 5% of total physical memory. Of the 267 MB used by Hound, 73 MB (27%) is used by age-ordered 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 heap pages by 47% (from 2726 pages to 1425 pages).

8.4 Staleness Computation

We evaluate Hound's data sampling technique for determining object staleness and compare it to the state-of-the-art code sampling technique used in SWAT. Hound and SWAT have opposing limitations: while SWAT can overestimate object staleness due to its sampling technique, it never underestimates staleness. Hound can underestimate object staleness since it tracks staleness at a page granularity, but it never overestimates staleness.

8.4.1 Accuracy Metrics

To quantitatively compare SWAT and Hound, we use several metrics. First, we use *precision* and *recall*, two metrics commonly used to measure the quality of classifiers in the information retrieval community. Precision is the ratio of true positives to all positives (true and false), while recall is the ratio of true positives to the sum of true positives and false negatives. Intuitively, classifiers with high precision (near 1.0) produce few false positives, and similarly, classifiers with high recall have low false negative rates. The reported precision and recall metrics are on a per-object basis, showing the accuracy of each approach at estimating individual object staleness.

However, leak detection tools generally do not report individual objects but rather aggregate them by their allocation callsite. In this context, object-based metrics can be misleading. Consider a report that identifies one allocation site as the source of stale data. If that report failed to identify nine other sites that also had stale data, the recall would be 0.1 (1/10). However, those nine unreported sites could have been responsible for only a tiny number of stale objects. For example, the reported site could be the source of 10,000,000 objects, and the nine unreported sites together might only comprise 100,000 objects.

To capture this effect, we introduce weighted metrics based on callsites rather than individual objects. *Weighted recall* weighs

each callsite by its *true* drag, summed over all objects allocated from that callsite. This weighting emphasizes the importance of false negatives that miss a significant volume of leaks or bloat. Similarly, *weighted precision* weighs callsites by their *reported* drag, emphasizing false positives that report larger amounts of (false) bloat.

8.4.2 Methodology

To compare Hound's precision and recall to SWAT, we implemented SWAT's Adaptive Bursty Tracing framework in PIN, a dynamic binary instrumentation system [15]. Our implementation instruments every memory reference and checks a flag to determine whether sampling is enabled. This approach sacrifices performance, but generates the same information as SWAT's adaptive sampling. We also use PIN to compute perfect staleness information which we use as ground truth when computing the precision and recall metrics for both Hound and SWAT.

For our results, we configure SWAT with a 0.1% sampling rate, as used in the original paper. While our experiments show the qualitative difference between SWAT and Hound with respect to accurate staleness information, computing precision and recall requires a binary classifier. We use SWAT's **Greater50M** predicate (stale for more than 50 million sampled memory references) to classify objects as leaks or non-leaks based on the staleness information from Hound, SWAT, or the oracle. Note that for this experiment, an object is considered leaked if it satisfies this predicate, regardless of whether or not it would be subjectively considered to be a leak.

8.4.3 Accuracy Results

Tables 1 and 2 present precision, recall, and weighted recall results for both Hound and SWAT across several benchmark programs. Note that in every case, Hound has a precision of 1.0 (no false positives), while SWAT has a recall of 1.0 (no false negatives).

HashTable is a microbenchmark that builds a hash table of two million objects and then executes 4,000 random probes. The hash table query code is hot, so it is rarely sampled. However, each individual data item is cold, since the table is large. Two allocation sites are hot: the one that produces objects stored in the table, and a site producing internal nodes. Note that none of these objects would be considered real leaks by a human, since each has an equal, non-zero probability of being accessed on the next query. SWAT's precision here is 50%, meaning that half of the objects it classifies as leaks are incorrect, because it overestimates staleness. Hound's recall is only 57% on this benchmark, meaning that it underreports staleness for 43% of individual objects. These results highlight the key difference between the two staleness tracking methods.

However, weighting reveals the qualitative difference between SWAT and Hound. Hound's weighted recall is 100%, meaning that it reports only the allocation site producing truly stale objects. By contrast, SWAT's weighted precision is 50%, because it reports both sites as stale.

Squid is a web cache application that acts 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 consults the indices of these caches to see whether it can satisfy the request locally, or must fetch the data from the hosting server. Squid 2.4STABLE3 and earlier suffer a memory leak when handling SNMP requests. We test Hound against this leak by sending a sequence of 20,000 requests with a mix of SNMP requests (leaks) and standard HTTP requests. SWAT's precision here is fairly high (88%), but that corresponds to 33 false alarms out of 549 allocation sites. Its weighted precision is 94%, in part because the heap has little false drag to report.

The GIMP is a graphic editor similar to Photoshop. We drive it with a script that automatically generates several images and then runs a series of effects and filters on them. SWAT's precision appears high (94%), but here, this value translates to 887 false positives out of 6,887 call sites. For this benchmark, Hound has a relatively low weighted recall (57%). We attribute this to the fact that the GIMP has many allocation sites with only a small number of objects per site, too few to cross Hound's tracking threshold.

Hound

	HashTable	Squid	Gimp
Precision (objects)	1.0	1.0	1.0
Weighted Precision (sites)	1.0	1.0	1.0
Recall (objects)	0.59	0.57	0.53
Weighted Recall (sites)	1.0	0.81	0.73

Table 1. Precision and recall metrics for Hound.

SWAT

	HashTable	Squid	Gimp
Precision (objects)	0.50	0.88	0.93
Weighted Precision (sites)	0.50	0.94	0.98
Recall (objects)	1.0	1.0	1.0
Weighted Recall (sites)	1.0	1.0	1.0

Table 2. Precision and recall metrics for SWAT.

8.5 Leak Identification and Ranking

We evaluated Hound's usefulness at isolating leaks and bloat qualitiatively by running it on Squid with the leak scenario described above. Figure 10 presents Hound's report. The report shows two callsites. The first is the true leaky allocation site in the `snmp_parse` function. Notice that its reported drag is an order of magnitude larger than the second site. This latter site eagerly creates data structures related to MIME type handling when the program starts. These data structures are never used during our test workload.

The information for the first site points the programmer directly to the exact source of the leaked objects. The second site indicates a possibly inefficient use of memory (i.e., bloat), which the programmer can use to determine the severity of the problem and make a subjective decision of whether to change the program's allocation behavior (e.g., by allocating this memory only when needed).

9. Related Work

Memory leaks and memory bloat have been the target of much previous work. Leak detection tools generally focus on two classes of errors: *unreachable* leaks that can be found by GC-based techniques, and staleness leaks which are reachable from live objects in the heap. Because Hound focuses on detecting staleness leaks, we focus our discussion of prior work on this area.

Dynamic leak detection

The prior work most closely related to Hound is SWAT [8], Sleigh [3], and SafeMem [24]. The relationship between SWAT and Hound has already been discussed. Sleigh is a leak detector for Java similar to SWAT. Sleigh also uses adaptive bursty tracing to reduce the overhead of staleness detection, and the authors report an increase in false positives as a result, although they still are able to detect a true leak amid noisy data. SafeMem uses ECC memory instead of program instrumentation to detect stale objects. It tracks allocation sites that appear to be leaking objects and uses ECC to reduce false positive rate for those sites. Unlike Hound, SafeMem cannot run on systems without ECC memory.

Other prior work focuses more on the issue of reachability. Conservative garbage collection techniques can be used to find

unreachable objects. Several tools use this approach, including Purify [7], Valgrind [21], and RADAR [18]. While these tools are useful for diagnosing a large class of leaks, they cannot find leaked objects that are still reachable.

Several papers focus on providing more detailed information about the causes of leaks to make debugging and correcting them easier. LeakBot automatically identifies Java data structures that are potential leaks by evaluating the evolving structure of the heap graph [19]. 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 [11]. Maebe et al. describe a high-overhead leak detector that identifies the specific program statement responsible for removing the last reference for reachability leaks [17]. Rayside and Mendel describe object ownership profiling, a high-overhead, trace-based dynamic technique that incorporates structural information when reporting leaks in object-oriented programs [25]. MemTracker provides state-tracking hardware support for individual memory locations, allowing low-overhead staleness detection [34].

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 that may indicate leaks [9]. Xie and Aiken use boolean constraints to find leaks based on escape analysis [36]. Cherem et al. propose an analysis that considers flows through the program graph from allocation points to deallocation points to identify possible leaks [5]. Orlovich and Rugina's analysis proves the absence of leaks, but can be used to detect leaks when the proof fails [23].

Leak tolerance

Several recent papers propose methods for leak *tolerance*, both in C/C++ and in garbage collected languages. For C applications, Cyclic Memory Allocation (CMA) [22] tolerates 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. It has not escaped our attention that Hound's virtual compaction scheme could be used to provide leak tolerance.

Garbage collection tolerates unreachable leaks by automatically reclaiming leaked objects, but does not address staleness leaks. Both Melt [4] and LeakSurvivor [31] augment a relocating garbage collector in Java with techniques to isolate and compress stale objects. Melt identifies staleness with a lightweight read barrier, while LeakSurvivor uses Sleigh's sampling mechanism. Tsai et al. combine statistical techniques with conservative garbage collection to detect and tolerate *unreachable* memory leaks in C applications [33].

Static analysis can also eliminate memory leaks by program transformation. Shaham et al. present two analyses that can eliminate memory leaks in Java: the first detects dead entries in arrays that will never be read in the future [28], while the second uses shape analysis to detect dead references [29]. Lattner and Adve propose pool allocation, a transformation that can statically eliminate some leaks in C/C++ applications via points-to set liveness [13].

VM-techniques for memory management

Appel and Li describe a number of primitives and algorithms for exploiting virtual memory in user-mode [1], including the primitives used by Hound. Dhurjati and Adve introduce a technique that uses virtual memory remapping to detect dangling pointer errors [6]. Each object is allocated on a new virtual page, with multiple virtual pages mapped to the same physical page. Their system detects dangling pointers by protecting the virtual pages holding freed objects. By contrast, virtual compaction starts with many objects mapped to individual virtual pages, and combines virtual pages (holding multiple objects) onto one 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. CRAMM is a virtual memory manager that provides detailed reference information, allowing it to dynamically adapt GC heap sizes in order to maximize performance [37]. Hound 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 [16] 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.

10. Conclusion

This paper presents Hound, a runtime system that precisely locates both memory leaks and sources of bloat. Hound's key contribution is its hybrid memory management scheme, which both segregates objects at allocation time with a *context-sensitive* allocator and separates leaked from non-leaked objects with an *age-segregated* allocator. A novel *virtual compaction* mechanism allows Hound to compact memory without the need to move objects, reducing fragmentation due to segregation without degrading Hound's ability to locate leaks.

Hound operates on unaltered binaries, making deployment simple. Hound locates both reachable and unreachable leaks without generating any false positives, and with extremely low overhead. For a range of applications, including servers and applications with low allocation-intensity, Hound incurs minimal runtime and memory overhead, making it practical for use even for large deployed applications where performance is a key concern.

11. Acknowledgments

The authors would like to thank Ting Yang and Scott Kaplan for valuable discussions about Hound and Linux virtual memory management, Ted Hart for his feedback during the development of Hound, and Shan Lu, Martin Rinard, and Huu Hai Nguyen for providing us the leaky inputs for *squid*. This material is based upon work supported by Intel, Microsoft Research, and the National Science Foundation under CAREER Award CNS-0347339 and CNS-0615211. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '91)*, pages 96–107, 1991.

[2] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '02)*, pages 1–12, 2002.

[3] M. D. Bond and K. S. McKinley. Bell: bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 61–72, San Jose, CA, Oct. 2006.

[4] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on*

Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008), pages 109–126, Nashville, TN, Oct. 2008. ACM.

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

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

[7] R. Hastings and B. Joyce. Fast detection of memory leaks and access errors. In *Proceedings of the Winter '92 USENIX conference*, pages 125–136. USENIX Association, 1992.

[8] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, pages 156–164, Boston, MA, Apr. 2004. ACM.

[9] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation (PLDI '03)*, pages 168–181, 2003.

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

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

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

[13] 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.

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

[15] 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 *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*, pages 190–200, 2005.

[16] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS '08)*, pages 115–124, Mar. 2008.

[17] J. Maebe, M. Ronsse, and K. D. Bosschere. Precise detection of memory leaks. In *Workshop on Dynamic Analysis (WODA 04)*, pages 25–31, 2004.

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

[19] 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.

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

[21] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[22] H. H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the 6th International Symposium on Memory Management (ISMM '07)*, pages 15–30, 2007.

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

[24] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*, volume 00, pages 291–302. IEEE Computer Society, 2005.

[25] D. Rayside and L. Mendel. Object ownership profiling: a technique for finding and fixing memory leaks. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering (ASE '07)*, pages 194–203, 2007.

[26] N. Röjemo and C. Runciman. Lag, drag, void, and use: Heap profiling and space-efficient compilation revisited. In *Proceedings of First International Conference on Functional Programming*, pages 34–41, Philadelphia, PA, May 1996. ACM Press.

[27] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*, pages 12–23, San Jose, CA, Oct. 1998.

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

[29] 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.

[30] Standard Performance Evaluation Corporation. SPEC2006. http://www.spec.org.

[31] Y. Tang, Q. Gao, and F. Qin. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX '08)*, pages 307–320, Boston, MA, June 2008.

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

[33] T. Tsai, K. Vaidyanathan, and K. C. Gross. Low-overhead runtime memory leak detection and recovery. In *Proceedings of the 12th Pacfic Rim International Symposium on Dependable Computing (PRDC '06)*, pages 329–340. IEEE Computer Society, 2006.

[34] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA '07)*, pages 273–284. IEEE Computer Society, 2007.

[35] 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, pages 1–116, Kinross, Scotland, Sept. 1995. Springer.

[36] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESAC/FSE '05)*, pages 115–125, 2005.

[37] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 103–116. USENIX Association, 2006.