Efficient Runtime Tracking of Allocation Sites in Java

Rei Odaira, Kazunori Ogata, Kiyokuni Kawachiya, Tamiya Onodera, Toshio Nakatani

IBM Research – Tokyo

1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan

{ odaira, ogatak, kawatiya, tonodera, nakatani } @jp.ibm.com

Abstract

Tracking the allocation site of every object at runtime is useful for reliable, optimized Java. To be used in production environments, the tracking must be accurate with minimal speed loss. Previous approaches suffer from performance degradation due to the additional field added to each object or track the allocation sites only probabilistically. We propose two novel approaches to track the allocation sites of every object in Java with only a 1.0% slowdown on average. Our first approach, the *Allocation-Site-as-a-Hash-code (ASH) Tracker*, encodes the allocation site ID of an object into the hash code field of its header by regarding the ID as part of the hash code. ASH Tracker avoids an excessive increase in hash code collisions by dynamically shrinking the bit-length of the ID as more and more objects are allocated at that site. For those Java VMs without the hash code field, our second approach, the *Allocation-Site-via-a-Class-pointer (ASC) Tracker*, makes the class pointer field in an object header refer to the allocation site structure of the object, which in turn points to the actual class structure. ASC Tracker mitigates the indirection overhead by constant-class-field duplication and allocation-site equality checks. While a previous approach of adding a 4-byte field caused up to 14.4% and an average 5% slowdown, both ASH and ASC Trackers incur at most a 2.0% and an average 1.0% loss. We demonstrate the usefulness of our low-overhead trackers by an allocation-site-aware memory leak detector and allocation-site-based pretenuring in generational GC. Our pretenuring achieved on average 1.8% and up to 11.8% speedups in SPECjvm2008.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – memory management (garbage collection)

General Terms Measurement, Performance, Reliability.

Keywords Memory allocation, hash code, allocation site

1. Introduction

Tracking the allocation sites where objects are allocated is useful to improve software reliability and performance. Many memory leak detectors [7,16,17,27,29] present a programmer with the allocation sites of possibly leaking objects. The allocation site information is also beneficial for debugging other memory-related problems such as buffer overflows [25,30]. Also, tracking allocation site enables various optimization techniques to be used. Allocation-site-based pretenuring [1,5,9,15,23] needs to collect statistics on the allocation sites of tenured objects in a generational garbage collector. Tracking allocation sites is also essential for other allocation-site-based optimizations such as object segregation [4,28], object co-allocation [10], reference locality optimization [11], and field-level optimization [8].

Unfortunately, previous methods of tracking allocation sites have serious limitations in production Java environments. Trace-based approaches [5,8,9,10,11,28] generate huge trace files for the allocation site each time an object is allocated. In practice, they can only be used for offline debugging and optimizations, limiting their capabilities compared to online methods. In contrast, many runtime tracking approaches [16,17,25,27,29,30] have per-object space overhead for the allocation site information. Some of them add an extra field to each object, which increases the CPU cache misses. The larger objects also increase the frequency and overhead of GC. Other approaches record per-object information in a separate table indexed by an object identifier, but this is not compatible with copying or generational GC because the object address cannot be used as the identifier. Several algorithms have been proposed to track allocation sites probabilistically, such as bit-encoding leak location [7] or object sampling [1,15,23]. However, they require a large number of samples to detect memory leaks, pretenuring opportunities, or other important events.

In this paper, we propose two novel techniques to track the allocation sites of every Java object at runtime with minimal speed overhead, which enables various kinds of online debugging and optimization in production environments. Our first technique, the *Allocation-Site-as-a-Hash-code (ASH) Tracker*, encodes an ID number for the allocation site of each object into part of its hash code field. In those environments where an object header contains the hash code field, including production-quality environments such as IBM J9 [14] and Sun HotSpot [33], ASH Tracker does not increase the per-object memory usage. A potential problem is that the hash code values should be as distinct as possible for distinct objects [21]. A novel idea in ASH Tracker is that it does not steal any bits from the hash code field. Instead it regards the allocation site ID as a part of the hash code field and puts an object-specific random value in the other bits. ASH Tracker avoids excessively increasing the hash code collisions by making the bit-length of the ID variable. ASH Tracker initially assigns long IDs to every allocation site. As an allocation site allocates many objects and as the hash code values of those objects are referred to, the site is assigned a new and shorter ID. As a result, those many objects can be distinguished from one another by leaving more bits in the hash code field for the random values.

For those environments where an object header does not contain a hash code field, we propose a second technique, the *Alloca-*

VEE'10 March 17--19, 2010, Pittsburgh, Pennsylvania, USA.

tion-Site-via-a-Class-pointer (ASC) Tracker. ASC Tracker uses the class pointer field of an object header to refer not to the class structure of the object but to its allocation site structure. The allocation site structure in turn points to the original class structure. Like ASH Tracker, ASC Tracker has no per-object space overhead. However, it can degrade performance because an additional load is required to access the class fields via the class pointer. To mitigate the indirection overhead, we developed two optimization techniques. One is to duplicate the frequently accessed almost-constant class fields, such as an instance size field, into each allocation site structure. The other is to avoid class equality checks in just-in-time (JIT) compiled code, which compare the class of an object with a pre-determined class to check the assumption of virtual-call inlining, the validity of class casting, and so on. The trick here is that allocation-site equality implies class equality. Thus we can replace the class equality checks with allocation-site equality checks. Combining these techniques, ASC Tracker achieves a far smaller slowdown than current approaches. ASC tracker suffers from slightly more overhead than ASH Tracker, but it can be applied to almost all Java environments because embedding the class pointer in each object header is a common implementation technique.

We demonstrate how useful our trackers are for both reliability and optimization purposes. For reliability, we implemented a minimal-overhead memory leak detector that records at each GC time the histories of the numbers of live objects on a per-allocation-site basis. Our leak detector can inform a user of the exact allocation sites of any leaking objects that appear even after running for days or weeks in a production environment. For optimization, we propose *Allocation-site-based Copy-time (A-C) Pretenuring* for generational GC. If certain allocation sites are found to frequently allocate long-lived objects, A-C Pretenuring pretenures the objects allocated at those sites. Instead of copying these objects many times in young semi-spaces, it copies them directly into a tenured space the first time they are copied during young GC. This can be implemented easily by modifying several program points in the GC, but it is not feasible without also tracking the allocation sites of every object at runtime.

The contributions of this paper are:

- We propose ASH and ASC Trackers to track the allocation sites of every object at runtime, which are the first techniques that enable various kinds of allocation-site-based online debugging and optimization in production environments.
- We implemented ASH Tracker and ASC Tracker in IBM J9/TR JVM [14]. We show their effectiveness through experiments with industrial standard benchmarks [12,13,31,32].
- We demonstrate the usefulness of our minimal-overhead trackers by an allocation-site-aware memory leak detector and by an allocation-site-based copy-time pretenuring.

The rest of the paper is structured as follows. Sections 2 and 3 explain the details of ASH Tracker and ASC Tracker, respectively. Section 4 presents experimental results. Section 5 shows the memory leak detector and A-C Pretenuring. Section 6 discusses related research and Section 7 concludes this paper.

2. Allocation Site as a Hash Code

This section describes ASH Tracker, a technique to encode an allocation site ID into a hash code field. Our principle is not to increase any per-object space overhead, so that ASH Tracker is robust in terms of space and speed even on applications that heavily use hash code. The allocation site of an object is the program site where the object is allocated, and its ID is a unique integer number assigned to the site. In fact, as explained in Section 2.2, ASH Tracker assigns multiple IDs to a single allocation site.

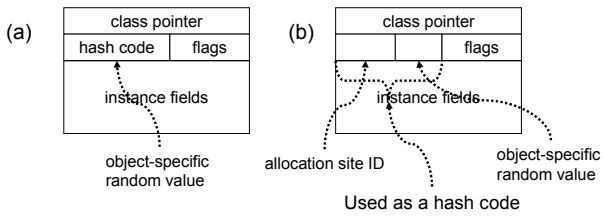

Figure 1. Object layout. (a) Original format. (b) ASH Tracker encodes an allocation site ID into a part of the hash code field.

2.1 Object layout

Throughout this paper, we assume Java is our target execution environment and the object shown in Figure 1(a) is the original object layout. However, the following discussion also applies to other environments that have a hash code or a class pointer in the object headers. The class pointer field points to the class structure of this object and is used to invoke virtual methods and to access per-class data. The flags are used by GC and a monitor, and the instance fields hold the actual data of the object. The hash code field contains the hash code value of the object, which should be as distinct as possible from all others. The field is usually initialized at allocation time with an object-specific random value, such as some of the address bits of the object. A program refers to the hash code value through standard interfaces such as `java.lang.Object.hashCode()` or `java.lang.System.identityHashCode()`. We call those objects whose hash code values are referred to *hashed* objects.

Figure 1(b) shows the object layout as modified by ASH Tracker. ASH Tracker encodes the allocation site ID of the object into a part of the hash code field. The bit position of the ID in the hash code field is insignificant. The rest of the bits are filled with an object-specific random value in the same manner as the original hash code field. ASH Tracker regards the bit combination of the ID and the random value as the hash code value of this object.

2.2 Algorithm

We use an artificial example in Figure 2 in this section. Actual data structures and pseudo code are explained in Section 2.4. This example allocates 600 objects (Loop A) and refers to the hash code values of every 10^{th} object (Loop B). Suppose the bit-length of the hash code field is six in this example. There are two allocation sites, i.e. two `new` expressions, in the example. For each allocation site, ASH Tracker assigns a *static ID* when an interpreter first executes the code at that site or when a JIT compiler compiles the method containing the site. When an object is allocated, the static ID of the site is embedded to it, occupying all of the bits of its hash code field. Since each allocation site always allocates objects of the same class and each object has a class pointer, it suffices to make the ID unique among the allocation sites of the same class. Figure 2(a) shows the hash code fields of the allocated objects immediately after Loop A. Note that we chose 14 representative objects among the 600. For example, since `x[0]` is allocated at P, 111111 is embedded into its hash code field.

When `hashCode()` is invoked on the objects in Loop B, their hash code values must be returned. However, simply returning the static IDs leads to excessive hash code collisions because all of the objects allocated at the same site contain the same static ID. Therefore, when `hashCode()` is invoked on an object for the first time, ASH Tracker replaces the embedded static ID with the current *dynamic ID* of its allocation site and its object-specific random value. If there is no current dynamic ID of the site, a new one is assigned. The return value of `hashCode()` is the combina-

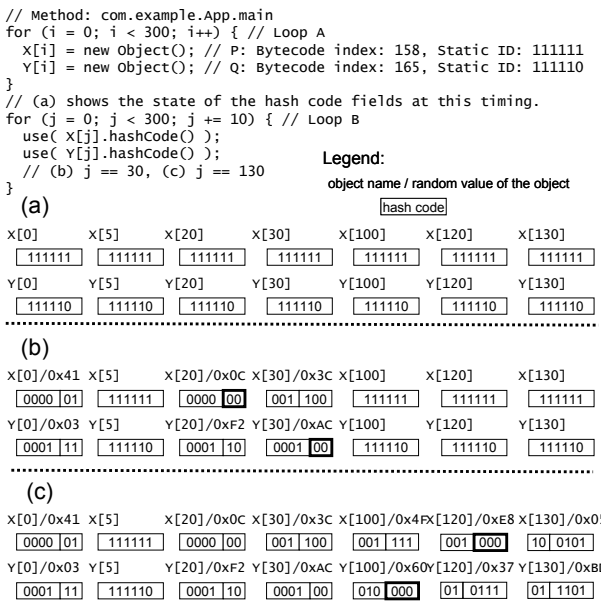

Figure 2. Example behavior of ASH Tracking. The program allocates 600 objects at two allocation sites, and refers to the hash code values of every 10th one. (a) The hash code fields of objects immediately after Loop A. We only show 14 among the 600 objects. Static IDs are set in the hash code fields. (b) During Loop B when `j==30`. `X[5]` and `Y[5]` still contains the static IDs, while `X[0, 20, 30], Y[0, 20, and 30]` have dynamic IDs plus their own random values. The random values of `X[20]` and `Y[30]` triggered dynamic shrinking for each site. (c) During Loop B when `j==130`. The random values of `Y[100]` and `X[120]` triggered further dynamic shrinking.

tion of the dynamic ID and the random value. Since the hash code value contains the random value, collisions can be avoided. Figure 2(b) shows the hash code fields during Loop B when `j` is 30. When `hashCode()` is invoked on `X[0]`, the allocation site 111111 is assigned a dynamic ID 0000. We also show the object-specific random value 0x41 of `X[0]`, which can be calculated from, for example, its address bits. The dynamic ID 0000 and the least significant two bits of 0x41 are embedded into the hash code field. On `Y[0].hashCode()`, a dynamic ID 0001 is assigned to the allocation site 111110. For `X[5]` and `Y[5]`, `hashCode()` is not invoked on them, so the hash code fields are unchanged.

Even if a random value is contained in a hash code value, the bit length of the random value may not be sufficient to avoid collisions if many objects are hashed. ASH Tracker solves this problem by assigning shorter dynamic IDs to an allocation site as more and more objects allocated at the site are hashed. Shorter IDs mean longer random values, which help avoid collisions. We call this process *dynamic shrinking*. However, it is not desirable to count the numbers of hashed objects because such global counters can become a major scalability bottleneck. Instead, ASH Tracker uses a probabilistic approach. A new shorter dynamic ID is assigned only when the bits of the random value field become all zero. An advantage of this approach is that the probability of the dynamic shrinking naturally becomes lower as the bit-length of the random value field becomes longer. Thus we can avoid consuming the bit space of short IDs too quickly. Section 2.5 explains an ID overflow. On `X[20].hashCode()` and `Y[30].hashCode()` in the example, the dynamic shrinking is triggered for each allocation site because the least significant two bits of their random values are all zero. Then the shorter dynamic IDs, 001 and 010, are assigned, respectively. Figure 2(c) shows the status in Loop B when `j` is 130. Further dynamic shrinking occurred at `Y[100]` and `X[120]`. Note that ASH Tracker avoids collisions by effectively distributing hash code values.

Once the hash code field of an object is set by `hashCode()`, the field is never modified again. When `hashCode()` is next invoked on the object, its hash code field is simply returned. To implement this behavior, we must be able to tell whether the field contains a static ID or a dynamic ID by looking at its bit pattern. For that purpose, ASH Tracker assigns static IDs from the top of the bit space of the hash code field, 111111 in this example, while the dynamic IDs are assigned from the bottom. It also maintains the current minimum static ID for each class, 111110 in this example. If the value in the hash code field of an object is smaller than the current minimum, the field contains a dynamic ID.

At any time during the execution, we can identify the allocation sites of all of the objects by inspecting their hash code fields. If the field contains a static ID, we simply consult a per-class table using the static ID as an index. If it contains a dynamic ID, we can find where the dynamic ID ends in the bit pattern by traversing down a per-class binary tree as explained in Section 2.4.

2.3 Rationale

The reason we embed a static ID instead of a dynamic ID at an allocation time is for smaller performance overhead. Embedding a static ID is lightweight especially when done with JIT-compiled code because the static ID is a compile-time constant. On the other hand, embedding a dynamic ID requires looking up the current dynamic ID. Since the number of hashed objects is usually smaller than the number of allocated objects [2], we defer embedding the dynamic ID until the object is really hashed.

The reason we do not assign a short dynamic ID from the beginning is that there are many allocation sites that allocate only a few objects, such as those on error-handling paths. We need long dynamic IDs to distinguish between those allocation sites. In contrast, there are only a couple of allocation sites that allocate many objects of a certain class [26]. Therefore, an overflow in ID numbers rarely occurs even with few bits.

2.4 Implementation

Figure 3 shows the data structures and pseudo code for ASH Tracker. The object type (`object_t`) represents the object layout shown in Figure 1. ASH Tracker maintains the current minimum static ID (`min_static_ID`) as described in Section 2.2. If the hash code value is smaller than the current minimum, then the value contains a dynamic ID, which we can simply return (Lines 4 and 5). A per-class lock (`cls_lock`) is acquired to perform dynamic shrinking (Lines 12 and 14). On the fastest path of `hashCode()`, one compare-and-swap is sufficient to replace the hash code field atomically (Line 16). A per-class random value (`cls_rand`) is needed to reduce hash code collisions among objects of different classes because the dynamic shrinking only helps avoid collisions among objects of the same class. The per-class random value is XORed with an ID, regardless of whether it is static or dynamic, and the resulting value is stored in the hash code field (Line 10). Thus, when decoding the hash code field, another XOR is needed (Line 3). When assigning a static ID, an interpreter or a JIT compiler consults a hash table (`map_table`) to map each allocation site (method + bytecode index) to an alloca-

```
struct object_t {
 class_t *cls_ptr;
 unsigned hash_code, flags;
 /* instance fields */ ……;
};

struct class_t {
 /* virtual function table,
  class variables, etc. */ ……;
 unsigned min_static_ID;
 mutex_lock_t cls_lock;
 unsigned cls_rand;
 alloc_site_t **map_table;
 alloc_site_t **site_array;
 dynamic_ID_node_t *tree_root;
};
```

```
struct alloc_site_t {
 method_t *method;
 bytecode_index index;
 unsigned static_ID;
 unsigned dynamic_ID;
 unsigned rand_mask;
 dynamic_ID_node_t *node;
};

struct dynamic_ID_node_t {
 alloc_site_t *alloc_site;
 dynamic_ID_node_t *parent,
  *child_zero, *child_one;
};
```

```
 1: int hashCode(object_t *this) {
 2:   RETRY: old_hash = this->hash_code;  cls = this->cls_ptr;
 3:   static_ID = old_hash ^ cls->cls_rand;
 4:   if (static_ID < cls->min_static_ID) {
 5:     return old_hash;
 6:   } else {
 7:     site = cls->site_array[static_ID];
 8:     new_hash = get_random_value(this) & site->rand_mask;
 9:     rand_val = new_hash;  new_hash |= site->dynamic_ID;
10:     new_hash ^= cls->cls_rand;
11:     if (rand_val == 0) {
12:       lock(cls->cls_lock);
13:       dynamic_shrinking(cls, site);
14:       unlock(cls->cls_lock);
15:     }
16:     if (COMPARE_AND_SWAP(&this->hash_code,
17:           old_hash, new_hash) == FAILURE) goto RETRY:
18:     return new_hash;
19:   }
20: }
```

Figure 3. Data structures and pseudo code for ASH Tracker.

tion site structure (`alloc_site_t`). A mapping from static IDs to allocation site structures is maintained in `site_array`.

Each allocation site structure records its containing method (`method`) and a bytecode index (`index`). This structure is first created when the corresponding static ID is assigned. The information can be enhanced by the dynamic calling context at the time the object is allocated. In this paper, we do not deal with dynamic calling contexts, but most optimizing JIT compilers perform aggressive method inlining, which has some of the same effects as using dynamic calling contexts. The current dynamic ID (`dynamic_ID`) and random value mask (`rand_mask`) are used to replace each static ID with the combination of the dynamic ID and a random value in `hashCode()` (Lines 8 and 9).

Dynamic shrinking is triggered when the bits of a random value are all zero (Line 11), as described in Section 2.2. By initializing `rand_mask` to zero, we can trigger shrinking at the first invocation of `hashCode()` for a site. Dynamic shrinking uses a binary tree rooted at a per-class `tree_root` to assign the dynamic IDs efficiently. Figure 4 shows an example that corresponds to the dynamic shrinking in Figure 2. Figure 4(a) shows a case when `hashCode()` is invoked on `X[0]`. Since this is the first time `hashCode()` is invoked on an object of `Object` class, the tree for `Object` class is created and initialized as shown. Dynamic ID 0000 is assigned to the allocation site P. The "(P)"s in the upper nodes indicate that these IDs can be assigned to P but are not yet assigned. The "--" in the root node means that it cannot be assigned to any site, because the static IDs (111111 and 111110) already implicitly occupy the same bit space. The "(P)" and "--" are only shown for this explanation. There is no need to set those values in the nodes. For `Y[0].hashCode()`, the initial ID 0001 is assigned to allocation site Q (Figure 4(b)). After that, none of the nodes from the root to 000 can be assigned to either site, because the allocation site of an object with 000 in the significant bits of its hash code could not be identified.

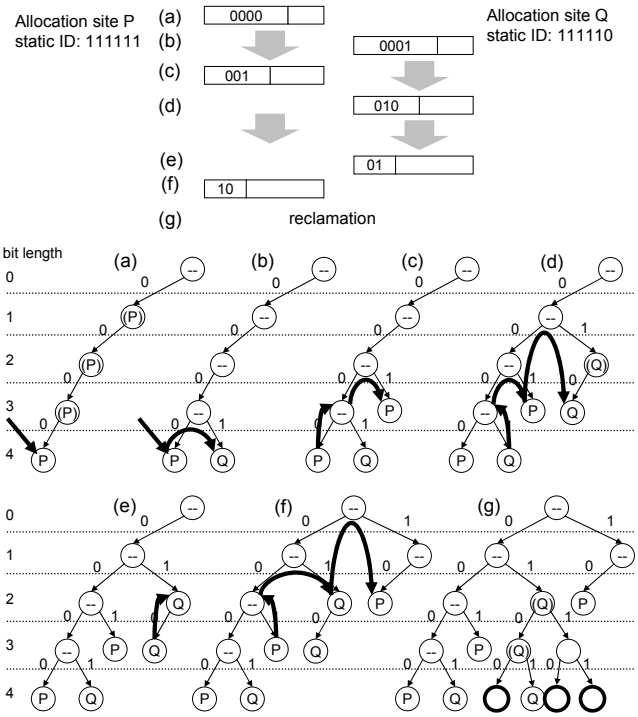

Figure 4. Assigning dynamic IDs using a binary tree. This example corresponds to the example in Figure 2. Each node in the tree is a dynamic ID, and the 0s and 1s on the edges from the root to the node represent its ID number. "P" or "Q" in a node indicates that the dynamic ID is already assigned to the allocation site. A node with "(P)" or "(Q)" is not yet assigned but is ready to be assigned to the site, while "--" means the ID cannot be assigned to any site. A thick arrow indicates a search for the next dynamic ID for an allocation site.

Here is the process for dynamic shrinking. On `X[20].hashCode()`, ASH Tracker searches for the next dynamic ID for P. The search starts from the current node and goes to its parent. The parent cannot be assigned to P because it has "Q" as a child, so the search continues to the node on the right side at the same height in the tree and a new node is created for P (Figure 4(c)). Figure 4(d) shows the search on `Y[30].hashCode()`. For `Y[100].hashCode()`, the parent node can be assigned to Q (Figure 4(e)). Finally, 10 is assigned to P in `X[120].hashCode()` (Figure 4(f)). Note that the dynamic ID 1 cannot be assigned to any allocation site because static IDs already occupy 111111 and 111110.

This binary tree also functions as the mapping table from dynamic IDs to the allocation sites. When decoding a dynamic ID, the bits of the dynamic ID are inspected from left to right to traverse down the tree accordingly. If a node assigned to an allocation site is encountered, that site is where the object was allocated. Note that using the binary tree, we can find where the dynamic ID ends in the bit pattern without explicitly having its bit-length.

2.5 Avoiding an overflow

Since the length of the hash code field is finite, an overflow in allocation site IDs is inherently possible. We want to avoid over-

flow whenever possible to make the allocation-site tracking as accurate as possible.

The most problematic overflow occurs when the maximum dynamic ID becomes equal to or greater than the minimum static ID. If this happens, ASH Tracker cannot assign a new static or dynamic ID to a new allocation site. To avoid this situation whenever possible, ASH Tracker monitors the coverage ratio of a tree relative to the bit space of the hash code. For example in Figure 4(f), the bit space is $2^6 = 64$. The dynamic IDs cover $2^2 + 2^2 + 2^3 + 2^4 + 2^4 = 48$, and there are two static IDs. Therefore, the coverage ratio is $(48 + 2) / 64 = 0.78125$. If the ratio exceeds a certain threshold, ASH Tracker will reclaim some of the dynamic IDs. Based on experiments, we chose 0.97 as the threshold for a 15-bit hash code field. The reclamation scans the whole Java heap and records the dynamic IDs embedded in live objects. ASH Tracker chooses the dynamic ID that is embedded in the smallest number of live objects. It then expands the sub-tree of the corresponding node in the tree. For example in Figure 4(g), suppose the node 01 is expanded and only the leaf 0101 has live objects. Then the other three leaves can be assigned to other allocation sites without compromising the accuracy of the tracking. The reclamation is lightweight because it is rarely invoked in practice and because the heap scanning can be piggybacked on the regular global GC.

ASH Tracker does not guarantee the uniqueness of allocation site IDs, but the successful reclamation keeps them unique as long as the number of allocation sites (= static IDs) plus the number of distinct hash code values being assigned to live objects are smaller than the bit space. Without the reclamation, the bit space for a class would get consumed proportionally to the number of hashed objects of the class from the beginning of a program's execution. Thus the overflow would eventually happen. With the reclamation, the occupied bit space is proportional to the number of *live* hashed objects. Thus the overflow will rarely happen even in a long running application.

The other overflow case is when dynamic shrinking cannot find a shorter dynamic ID for an allocation site. This can increase hash code collisions but does not affect the accuracy of allocation site tracking. Therefore, the allocation site simply continues to use the current dynamic ID.

3. Allocation Site via an Indirect Class Pointer

This section describes ASC Tracker, a technique to track allocation sites when an object does not contain a hash code field.

3.1 Organization

Figure 5(a) shows the original components and the links from objects to their class structures. The class pointer field in each object header points directly to the class structure and is used to access per-class fields. For example, GC finds the size of a non-array object by looking at the instance size field of its class structure. A virtual function table to implement polymorphism is also normally located in the class structure. In addition, the class pointer field of an object is often used in class equality checks. For example, the field is compared against another class pointer value to check the assumption of an inlined virtual call.

ASC Tracker inserts an allocation site structure between an object and a class structure, as shown in Figure 5(b). Objects allocated at the same site point to their allocation site structure, which in turn points to the actual class structure. This organization is possible because any one allocation site always allocates objects of the same class. An additional load instruction is necessary to access the class fields and to perform the class equality check. The allocation site of an object can obviously be identified by dereferencing its indirect class pointer field. Like a static ID in ASH Tracker, an allocation site structure is created when an interpreter first executes the code at that site or when a JIT compiler compiles the method containing the site.

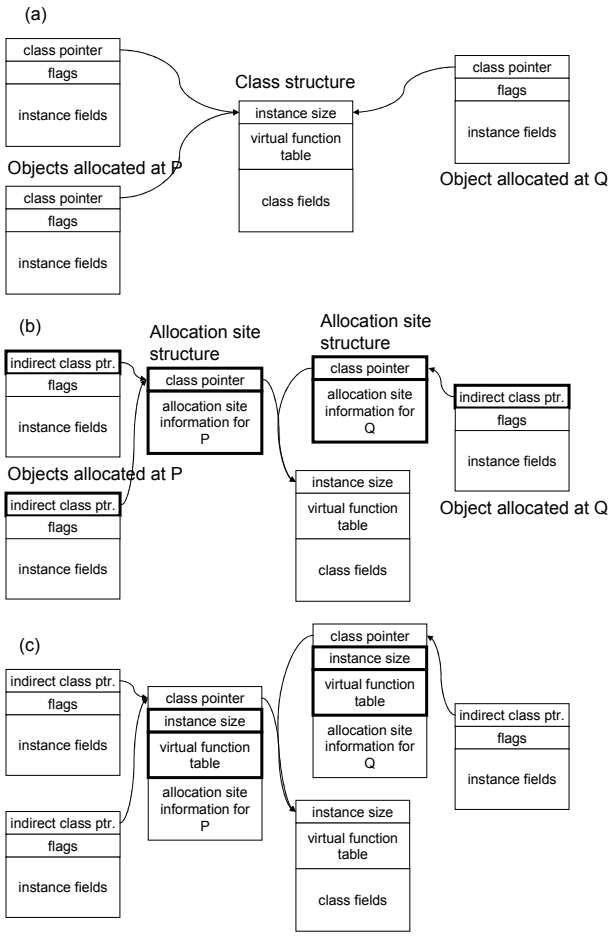

Figure 5. (a) Original components. A class pointer field points directly at a class structure. (b) ASC Tracker makes the class pointer field point to an allocation site structure. (c) Mostly unchanging fields can be duplicated to each allocation site structure to mitigate the indirection overhead.

3.2 Optimizations

To reduce the indirection overhead, ASC Tracker implements two types of optimizations. One is constant-class-field duplication and the other is allocation-site equality checks.

Constant-class-field duplication

Most of the class fields that are frequently accessed via the class pointer field of an object are found to be constant or rarely changed. ASC Tracker duplicates such fields for each allocation site structure as shown in Figure 5(c), so that they can be accessed with the same number of load instructions as before. For example, GC needs the instance size field, which is constant during a program's execution. As another example, the element class of an array class is also constant. Java requires class checking at each store to an array of references [22], which accesses the element class of the array class of the destination array object. A virtual

```
(a) if (object->cls_ptr != HOT_CLASS)
      goto SlowPath;
    /* Fast path here, such as an inlined call, assumes
       that object is an instance of HOT_CLASS. */

(b) if (object->alloc_site->cls_ptr != HOT_CLASS)
      goto SlowPath;

(c) if (object->alloc_site != HOT_ALLOCATION_SITE)
      goto SlowPath;
    /* Fast path here assumes that object is
       an instance of HOT_ALLOCATION_SITE->cls_ptr. */
```

Figure 6. (a) Class equality check. (b) ASC Tracker requires another load instruction. (c) Allocation-site equality check reduces the number of load instructions to the same as for (a).

function table is also a candidate for the duplication. A JIT compiler occasionally installs the entry of a JIT-compiled method into multiple table slots. This installation need not be done atomically, because JIT compilation is not a semantic change but just an optimization. Of course, the class-field duplication is at the cost of space and cache misses. ASC Tracker must carefully choose which class fields to be duplicated. As we will describe in Section 4.1, we chose to duplicate the instance size and element class fields. Also note that class fields visible from Java programmers, such as `System.out`, are not accessed via an object. Therefore there is no indirection overhead for them.

Allocation-site equality checks

Optimizing JIT compilers generate many class equality checks. Figure 6(a) shows a typical one, which compares the class pointer of an object with a compile-time-constant class pointer value, using a fast path if the check succeeds. This kind of checking often appears when a JIT compiler inlines a virtual call. If profiling reveals that a certain class accounts for most of the receiver classes of the virtual call invocation, the compiler can inline the call by assuming the receiver is an instance of the dominant class and can generate an class equality check before the inlined code. The same kind of checking is also used in the fastest path of a class cast check (`checkcast/instanceof`).

Without optimization, ASC Tracker requires another load instruction for the class equality check as shown in Figure 6(b). However, if we find that a certain allocation site accounts for most of the allocation sites of the objects, we can generate an equality check for the allocation site as in Figure 6(c). Since an allocation site always allocates objects of the same class, we can assume that on the fast path, the object is an instance of the class allocated at the site. We call this check an *allocation-site equality check*. ASC Tracker profiles the allocation sites of receiver objects at virtual calls and argument objects at class cast checks. If it finds a certain allocation site is dominant, it generates an allocation-site equality check. If not, it generates a class equality check.

4. Experiments

This section describes our implementation of ASH and ASC Trackers and our experiments to evaluate their effectiveness.

4.1 Implementation

We implemented ASH and ASC Trackers in 64-bit IBM J9/TR 1.6.0 SR3 [14]. An object header consists of three words, each of which is 32 bits long because our virtual machine supports compressed pointers. We used 15 bits in the object header as the hash code field. The baseline implementation embeds 15 bits from the object address, while the ASH Tracker embeds a 15-bit static ID. For the dynamic shrinking, the initial bit-length of a dynamic ID is 11 and the minimum is 2. This implementation effectively avoids an overflow.

Our ASC Tracker duplicates to each allocation site structure the instance size field of a non-array class and the element class field of an array class. We did not duplicate a virtual function table because experiments showed that it increased CPU cache misses. The ASC Tracker generates allocation-site equality checks at inlined virtual method calls, interface calls, and class cast checks, based on value profiling.

We also implemented an earlier approach, which adds an extra field to each object. Our implementation appends an additional 4-byte field at the end of each object. Since objects are aligned with at least a 4-byte boundary, it is a waste of space to add a field of shorter than 4 bytes. The extra field contains a pointer to the allocation site structure of the object. The allocation site structures are allocated in a low region of the memory space, so that they can be addressed using a 32-bit pointer.

Our JVM first executes a method using an interpreter, and if the method is executed frequently, it invokes JIT compilation. As the method is executed more often, our JVM can invoke recompilation multiple times to apply more advanced optimizations, including aggressive method inlining and redundancy elimination.

Our virtual machine is equipped with a two-generation GC, and we used the following configuration [19]. One fourth of the Java heap is used as the young space, which consists of two semi-spaces. The young GC copies an object multiple times between the semi-spaces before making it tenured. The age of an object is stored in its header. The global GC performs mostly concurrent mark-and-sweep collection. These are the default configurations in our JVM, which has been tested on multiple customer applications in production environments. Similar configurations of GC have been adopted also in Sun HotSpot [33]. Our implementation of the ASH Tracker requests the reclamation of dynamic IDs when the coverage ratio for a class described in Section 2.5 exceeds 0.97. We chose this number based on experiments. The actual reclamation is performed during the next global GC.

4.2 Benchmarks and evaluation environment

To evaluate the effectiveness of our techniques, we used SPECjvm2008 [32], DaCapo Benchmarks [12], SPECjbb2005 [31], and Eclipse IDE for Java EE Developers 3.4.2 [13]. We excluded the scimark and crypto benchmarks from SPECjvm2008 because they create fewer objects than the other benchmarks and thus are less interesting for allocation site tracking. We did not use bloat, jython, and xalan from DaCapo Benchmarks because their execution times did not converge within 20 iterations and thus are not reliable. We used a single-JVM configuration for SPECjbb2005. For Eclipse, we created a new project for Java and another new project for plug-in development, imported the source files of SPECjbb2005 as the Java project, and then manually opened all of the files. We call this benchmark "Eclipse/manual" because DaCapo Benchmarks also include another eclipse benchmark. We ran the benchmarks eight times by restarting the JVM each time and took the averages. For SPECjvm2008, we performed a two-minute warm-up and a four-minute iteration. For DaCapo Benchmarks, we used large data sets and ran each benchmark with at most 20 iterations until the execution time converged for 5 consecutive iterations. We ran the benchmarks on an 8-core 1.8 GHz Intel Xeon [20] machine with 8GB of main memory running Linux 2.6.18. We measured the minimum Java heap requirement for each benchmark, and specified four times the minimum as the initial and maximum Java heap sizes.

	4x the minimum Java heap (MB)	Avg. physical memory (MB)	Number of allocated objects	Avg. number of total (live + dead) objects	Avg. number of live objects	Avg. hashed (%)	Number of times hash codes are referred to	Avg. copied / tenured (%)
compiler.compiler	816	915	4.80 * 100,000,000	1.83 * 10,000,000	6.33 * 1,000,000	0.70	3.34 * 100,000,000	16.2/14.6
compiler.sunflow	508	660	1.20 * 1,000,000,000	1.18 * 10,000,000	3.75 * 1,000,000	0.36	2.87 * 100,000,000	21.7/20.0
compress	384	485	5.74 * 100,000	7.04 * 10,000	4.78 * 10,000	0.22	617	8.8/5.1
derby	2032	2214	8.20 * 1,000,000,000	5.36 * 10,000,000	6.29 * 1,000,000	0.71	1.57 * 10,000,000	0.04/0.02
mpegaudio	36	111	5.77 * 1,000,000	4.93 * 10,000	4.05 * 10,000	0.34	780	5.8/1.5
serial	1220	1437	3.09 * 1,000,000,000	2.62 * 10,000,000	9.98 * 10,000	5.30	1.02 * 1,000,000,000	0.06/0.001
sunflow	60	143	2.60 * 1,000,000,000	1.34 * 1,000,000	2.42 * 100,000	0.19	2.72 * 10,000	1.2/0.82
xml.transform	164	315	2.87 * 100,000,000	2.23 * 1,000,000	2.32 * 100,000	0.20	5.62 * 1,000,000	3.0/0.33
xml.validation	252	340	6.42 * 100,000,000	3.37 * 1,000,000	1.92 * 100,000	0.07	2.02 * 100,000,000	9.0/1.4
antlr	12	71	1.01 * 10,000,000	1.81 * 100,000	4.25 * 10,000	0.61	8.49 * 1,000	6.5/2.6
chart	80	237	9.20 * 10,000,000	2.18 * 1,000,000	2.75 * 100,000	0.12	3.17 * 1,000	5.1/0.36
eclipse	160	257	1.39 * 100,000,000	2.37 * 1,000,000	5.34 * 100,000	1.08	2.01 * 10,000,000	6.8/1.8
fop	44	82	1.19 * 1,000,000	1.02 * 1,000,000	2.57 * 100,000	0.09	357	14.4/13.9
hsqldb	832	604	1.16 * 10,000,000	8.80 * 1,000,000	6.67 * 1,000,000	0.004	2.14 * 10,000	5.7/4.9
luindex	12	82	1.16 * 10,000,000	2.84 * 100,000	6.23 * 10,000	0.38	1.19 * 1,000	6.5/0.23
lusearch	36	195	4.60 * 10,000,000	3.95 * 100,000	6.68 * 10,000	0.40	549	0.56/0.14
pmd	76	165	1.54 * 100,000,000	2.47 * 1,000,000	2.63 * 100,000	11.61	3.06 * 10,000,000	2.3/0.57
SPECjbb2005	2176	2340	1.20 * 10,000,000,000	4.46 * 10,000,000	1.13 * 10,000,000	0.003	5.53 * 1,000	2.1/0.08
Eclipse/manual	400	637	2.92 * 10,000,000	4.37 * 1,000,000	1.44 * 1,000,000	0.79	1.31 * 1,000,000	9.0/5.0

Table 1. General characteristics of the benchmarks.

	Number of classes	Number of allocation sites	Number of allocation sites in inlined contexts	Number of allocation sites with dynamic shrinking	Max static ID	Frequency of ID reclamation (times/min)	Frequency of global GC (times/min)
compiler.compiler	1,491	11,475	2,658	261	1,752	1.2	11.7
compiler.sunflow	1,481	11,095	2,356	254	1,698	1.8	23.5
compress	1,174	7,311	226	176	1,278	0	0.3
derby	1,619	11,198	1,076	202	1,352	0	0.3
mpegaudio	1,205	10,587	270	177	3,327	0	4.5
serial	1,239	7,943	472	239	1,323	0	0.3
sunflow	1,257	8,242	364	186	1,504	0	33
xml.transform	2,221	18,964	2,834	438	1,887	0.2	6.3
xml.validation	1,412	9,329	626	187	1,501	0	5.0
antlr	6,68	6,852	1,462	145	1,319	0.1	11.6
chart	1,109	8,655	1,081	156	1,343	0	0.6
eclipse	1,713	14,006	3,623	352	1,421	0.6	1.8
fop	1,082	4,965	351	143	560	0	13.6
hsqldb	707	4,351	387	147	551	0	41.7
luindex	665	4,326	657	143	602	0	7.3
lusearch	658	4,226	645	143	599	0	18.9
pmd	832	4,446	567	227	554	0.2	4.2
SPECjbb2005	981	6,269	373	176	984	0	0.0008
Eclipse/manual	6,821	29,107	3,108	746	1,459	0	7.3

Table 2. Statistics on the allocation sites in the benchmarks.

Table 1 summarizes the general characteristics of the benchmarks we used. The second column shows four times the minimum Java heap requirement for each benchmark, as we specified to our JVM. The third column indicates the average physical memory usage during the steady state of each benchmark. This includes Java heap, Java classes, static code, JIT-generated code, and JVM-internal data structures. The usage can be smaller than the specified Java heap size because OS does not necessarily assign physical memory to the entire Java heap. The fourth column is the number of allocated objects during one iteration of each benchmark. The fifth column is the average numbers of objects before GC, while the sixth column is live objects after GC. The seventh column is the percentage of the live objects on which `java.lang.Object.hashCode()` or `java.lang.System.identityHashCode()` was invoked at least once. We also counted the number of times those standard interfaces were invoked during one iteration of each benchmark, as shown in the eighth column. Note that we did not count the invocations on the objects of sub-classes that overwrite `hashCode()`. The results indicate that the usage of the hash code value in the object header greatly varies from program to program. The last column shows the percentages of the allocated objects that were copied at least once or tenured during young GC.

4.3 Allocation site statistics

Table 2 shows the statistics on the allocation sites for each benchmark. The second column is the number of classes at least one of its allocation sites was assigned a static ID in ASH Tracker. The third column is the total number of allocation sites for all of the classes. The fourth column shows the total number of the allocation sites that appeared in inlined contexts during JIT compilation. Although our JIT compiler performs aggressive method inlining, only less than 26% of the allocation sites were in inlined

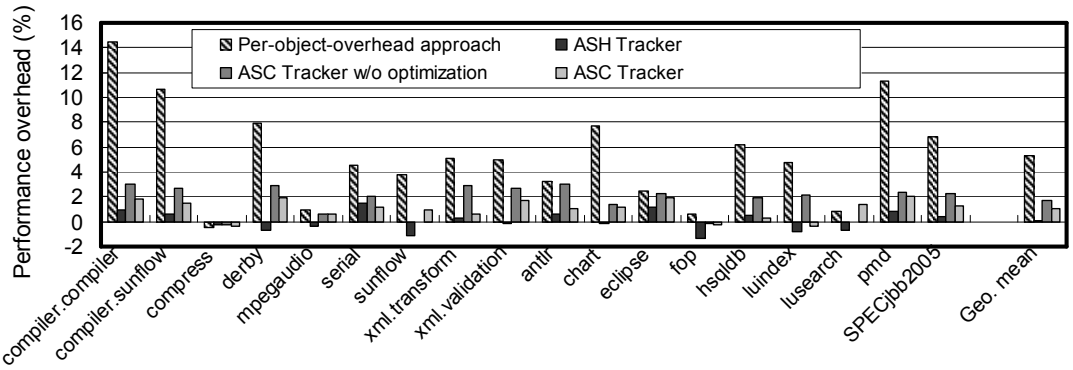

Figure 7. Relative performance overhead compared with the baseline that does not track allocation sites.

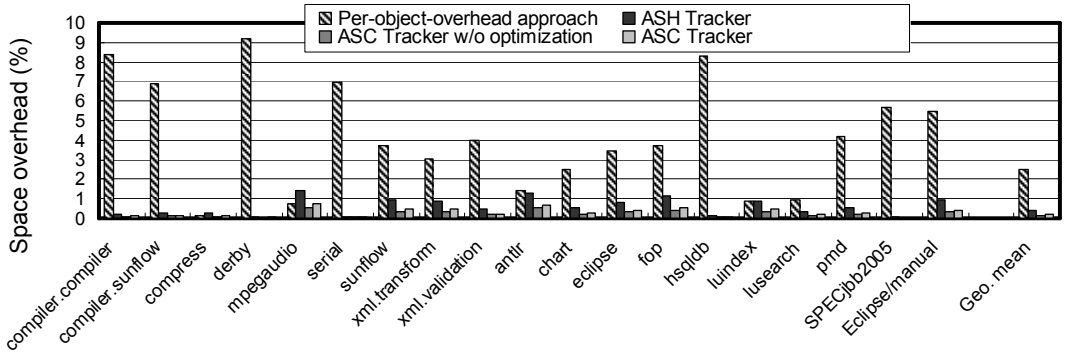

Figure 8. Relative space overhead compared with the physical memory usage. The per-object-overhead approach suffers from 4 bytes of per-object overhead times the average number of objects. The overhead in ASH Tracker mainly consists of per-class arrays and binary trees. The overhead in ASC Tracker includes duplicated constant class fields.

methods. These results suggest future work on embedding a dynamic allocation context into an object header. The fifth column is the total number of the allocation sites to which dynamic shrinking was applied in ASH Tracker, which means the allocation sites that allocated at least one hashed object. Since the value is less than 5% of the number of allocation sites, the space overhead for the binary trees is negligible in practice as explained in Section 4.5. The sixth column is the maximum static IDs assigned to an allocation site of a class in each program, which is the maximum number of allocation sites per class. Since the maximum was 3,327, all of these IDs were successfully embedded into the 15-bit hash code field in ASH Tracker. We confirmed that no ID overflow due to a conflict between the bit spaces of the static and dynamic IDs occurred in any of these benchmarks. The seventh and eighth columns are the frequency of the reclamation of dynamic IDs and global GC, respectively. These columns show that the reclamation during global GC is rarely triggered in practice.

4.4 Performance overhead

We compared the performance overhead of the per-object-overhead approach, ASH, and ASC Trackers with the baseline implementation that does not track allocation sites. We exclude Eclipse because we controlled it manually in the experiments. The results are reported in Figure 7. The bars are, from left to right, the per-object-overhead approach, ASH Tracker, ASC Tracker without optimizations, and ASC Tracker with optimizations.

The per-object-overhead approach slowed down the benchmarks by up to 14.4% and 5.2% on average because an additional field for each object increased the CPU cache misses, GC frequency, and copying GC's overhead. In compiler.compiler, the cache misses in Java code increased by 18%, and 24% more time was spent in GC. For ASH Tracker, the overhead was negligible on average and up to 1.4% in serial. ASH Tracker can degrade performance if many objects are hashed. At the same time, it can speed up object allocation because embedding a static ID, which is a compile-time constant value, is faster than computing a 15-bit random value from an object address.

ASC Tracker without any optimizations suffered from at most 3.0% and on average 1.8% overhead. The extra overhead for virtual calls was mitigated by aggressive method inlining. In addition, redundancy elimination removed unnecessary load instructions for class and allocation-site pointers. Our optimizations further reduced the overhead to 2.0% at maximum and 1.0% on average. The allocation-site equality checks reduced more overhead than the constant-class-field duplication in most benchmarks because there were many class equality checks in the frequently executed paths of JIT-compiled methods. Overall, ASH Tracker and ASC Tracker incurred far less performance overhead than the per-object-overhead approach in the benchmarks.

4.5 Space overhead

Figure 8 shows the relative space overhead, compared with the average physical memory usage shown in the third column of Table 1. The overhead of the per-object approach consists of per-class, per-allocation-site, and per-object structures. The first two structures are base overhead, which is unavoidable when tracking

	Baseline (15 bits from address)		ASH Tracker	
	Time avg. of avg. collision	Time avg. of max collision	Time avg. of avg. collision	Time avg. of max collision
compiler.compiler	1.921	12.757	3.328	15.149
compiler.sunflow	1.622	9.119	1.654	10.979
compress	1.000	1.000	1.000	1.000
derby	1.898	13.684	2.222	20.446
mpegaudio	1.007	1.999	1.007	1.999
serial	1.534	13.958	1.712	10.968
sunflow	1.054	3.999	1.066	3.999
xml.transform	1.084	2.045	1.105	3.437
xml.validation	1.015	1.997	1.015	1.999
antlr	1.005	1.688	1.065	3.785
chart	1.010	2.023	1.096	4.986
eclipse	1.849	6.713	2.014	20.163
fop	1.004	1.901	1.062	3.702
hsqldb	1.009	1.810	1.034	3.759
luindex	1.004	1.995	1.119	3.988
lusearch	1.004	2.000	1.074	3.000
pmd	3.178	12.791	4.111	19.640
SPECjbb2005	1.005	1.995	1.027	3.998
Eclipse/manual	1.185	4.000	1.902	13.500

Table 3. Time average of the average and maximum collision of hash code values.

allocation sites. They correspond to `class_t::map_table`, `alloc_sit_t::method`, and `alloc_site_t::index` in Figure 3. Their overhead was less than 1%, so most of the overhead was due to per-object structures. The overall overhead reached 9.2% in derby and 8.3% in compiler.compiler. These benchmarks used many objects during their executions, as shown in the fifth column of Table 1.

In contrast, ASH Tracker incurred much smaller space overhead of less than 1.4%. The space overhead specific to ASH Tracker consists of the per-class array indexed by a static ID (`class_t::site_array` in Figure 3) and the binary trees used for dynamic ID assignment. Since they are allocated separately from each object, they do not pollute the CPU caches unlike a per-object extra field. The space overhead of ASC Tracker is even smaller, less than 0.7%. Its overhead is the sum of the per-class and per-allocation site base overhead and duplicated constant class fields. The duplicated fields are 8 bytes per allocation site, so that their overhead was negligible.

4.6 Hash code collisions

Table 3 presents the results of hash code collisions in the baseline implementation, which uses 15 bits from an object address, and our ASH Tracker. In this experiment, we only took account of the hashed objects, which corresponds to the seventh column in Table 1. After every GC, we scanned the Java heap and computed the average number of collisions by dividing the number of such objects by the number of distinct hash code values. If every object has a distinct hash code value, the average collision number should be one. We also recorded the maximum number of collisions at that time. After execution was finished, we computed the time averages of those average and maximum collision numbers.

The average collision number in ASH Tracker was within 1.73 times the baseline, although the maximum collision number was up to 3.38 times larger. The collision number became large when dynamic shrinking chose the parent node of the current node. The case in Figure 4(e) is an example. In that example, hash code values with the dynamic ID 010 will appear frequently, because about half of the hash code values with the dynamic ID 01 will also have 010 in their significant bits. This is a tradeoff between an ID overflow and hash code collision. We currently prefer to avoid an overflow because it compromises tracking accuracy.

4.7 Discussion

Among the three approaches we compared, ASH Tracker achieved the least performance overhead. If an object contains a hash code field whose bit-length is long enough to encode allocation site IDs, ASH Tracker is the first choice. If not, ASC Tracker provides the tracking functionality with sufficiently small overhead. For accurate tracking, the hash code field must be long enough at least to contain static IDs. The sixth column of Table 2 indicates that 12 bits are sufficient for these benchmarks.

ASC Tracker can also be used for tracking other information such as last-use points. Each time an object is used, the pointer to a use-point structure of the class of the object is looked up and stored in the class pointer field. ASH Tracker is suitable only for encoding allocation-time constants because the hash code value of an object is not allowed to change once it is referred to [21].

For reflective calls like `Class.newInstance`, we used "anonymous" per-class allocation site structures in our experiments. Instead, we could track the allocation sites by passing an allocation site ID to a generic allocation routine in the JVM.

For static language environments without GC like C++, ASC Tracker is a suitable approach. Allocation site structures can be statically generated in the data sections of object files.

5. Applications of Allocation Site Tracker

We demonstrate the usefulness of our minimal-overhead allocation site trackers by implementing a memory leak detector and object pretenuring.

5.1 Minimal-overhead Memory Leak Detector

One of the most obvious applications of an allocation site tracker is a memory leak detector that tells users where leaking objects were allocated. Since Java is equipped with GC, a leak is a situation where the number of live objects increases steadily. With the allocation site information, the users can focus on a particular program region to fix the leak. Information about the classes of the leaking objects is not sufficient. For example, the `String` class may have 1000 allocation sites in typical programs.

It is important to be able to always enable the memory leak detector while an application is running. This is because users need to be able to investigate the histories of the number of live objects on a per-allocation-site basis to find the objects that are causing leaks. A leak can appear after a production environment has been running days or weeks [16]. Thus any slowdown from the memory leak detector must be as small as possible. If performance suffers, many users will prefer to disable the detector in production environments, even if there are risks of reduced reliability.

We implemented a memory leak detector that works with either ASH or ASC Tracker. Each time global GC occurs, the detector scans the Java heap and counts the numbers of live objects for each allocation site. It records the numbers in per-allocation-site areas and also saves exponential histories, which are the numbers of live objects at the most recent global GC, at the second most recent, at the fourth most recent, at the eighth most recent, and so on. Our implementation saves 16 histories for each allocation site. At any time, users can look at the histories of the allocation sites to detect leaking allocation sites. Also, the system can automatically inform users about the possible memory leaks. To define exactly which situations should be regarded as memory

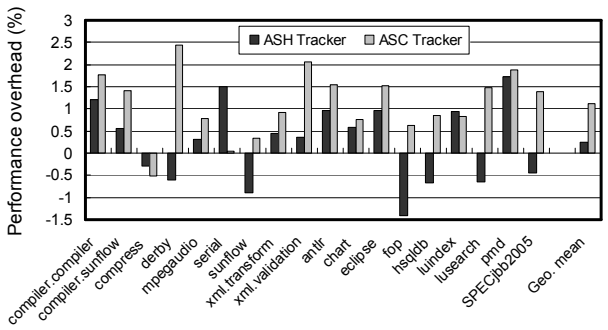

Figure 9. Relative slowdown of our memory leak detector compared with the baseline with no memory leak detection or allocation site tracking.

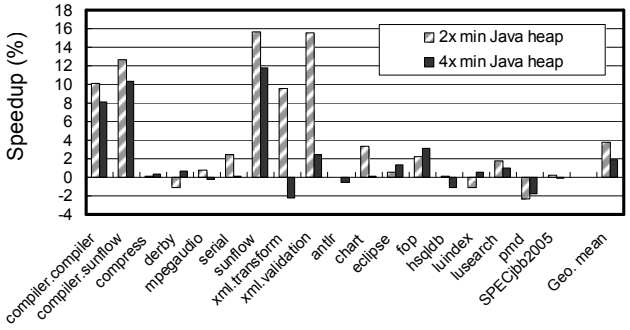

Figure 10. Performance of A-C Pretenuring on ASH Tracker with two different Java heap sizes. The baseline is no A-C Pretenuring or ASH Tracker.

leaks is beyond the scope of this paper, but our memory leak detector provides all of the necessary information for decisions.

Figure 9 shows the slowdown caused by our memory leak detector. The baseline is no memory leak detection or allocation site tracking. The overhead includes not only that of the tracking but also that of identifying the allocation sites for each of the live objects at each global GC. The slowdown was a maximum of 2.4% in derby when using ASC Tracker, and on average 0.2% for ASH Tracker and 1.1% for ASC Tracker. Identifying the allocation sites of objects slowed performance by only 0.1 to 0.2 % on average. Decoding a dynamic ID in ASH Tracker requires traversing down a binary tree, which caused the slowdown for pmd. The results show that ASH and ASC Trackers provide an efficient infrastructure for our minimal-overhead memory leak detector.

5.2 Allocation-site-based runtime pretenuring

Tracking the allocation sites of every object with minimal overhead is also indispensable for allocation-site-based optimizations. We demonstrate it by implementing allocation-site-based runtime pretenuring in our generational GC framework. Pretenuring is an optimization that allocates or moves a likely-to-be-long-lived object directly into the tenured space. Pretenuring can save the overhead of young GC because otherwise the young GC would copy the object multiple times within the young space before making it tenured. Since the pretenuring decision should be adaptive to application behavior, there have been several approaches proposed for runtime pretenuring [1,15,23]. In this paper, we assume two generations, young and tenured, but the same discussion applies to a collector with more than two generations.

Allocation-site-based Copy-time (A-C) Pretenuring

Pretenuring is known to be effective when it is performed on a per-allocation-site basis [5,9], which means that an object gets pretenured at a certain time during its lifetime if it was allocated at a certain allocation site. Previous allocation-site-based approaches perform pretenuring at the object allocation time. If an allocation site is found to allocate many long-lived objects via profiling, the site is modified to allocate objects directly into the tenured space. A problem in the allocation-time approach when it is performed at runtime is that it requires modifications to the code generation in the JIT compiler [1,15,23]. This is because current optimizing JIT compilers inline the fastest path of object allocation in the JIT-generated code.

We propose an easier-to-implement approach, which we call Allocation-site-based Copy-time (A-C) Pretenuring. The copy-time approach performs pretenuring during young GC when an object is copied for the first time. A generational collector usually has logic written in a high-level language to decide whether an object should be tenured or not based on its age. Thus the copy-time approach does not need to modify the JIT-generated code. A developer can write the pretenuring logic in a high-level language. A copy-time approach was described in a previous report [1], but it was not allocation-site-based. The allocation-site-based copy-time approach is impossible without tracking allocation sites of every object at runtime like ASH Tracker.

A drawback of the copy-time approach is that it must copy any object at least once. However, it also has an advantage in that it can filter out the lifetime heterogeneity of an allocation site. Even if an allocation site allocates short-lived objects that die before the next young GC mixed with other longer-lived objects, our copy-time approach can effectively apply pretenuring to the site.

Online profiling

Pretenuring in general relies on profiling to decide which objects should be pretenured. In A-C Pretenuring, when an object is copied for the first time during young GC, a counter `s` associated with the allocation site of the object is incremented. When the object is tenured later, another counter `t` for the allocation site is incremented. If the ratio of `t/s`, which we call the tenuring ratio, exceeds a certain threshold, pretenuring is turned on for that allocation site. In our implementation, the counters are saved and cleared each time young GC is invoked. Since our young GC copies an object multiple times in the young space before making it tenured, there is a time lag between the increments of `s` and `t` for a particular object. Thus we preserve the last 16 values of both of these counters and enable pretenuring when the tenuring ratio of the average of these 16 historic values exceeds a threshold. We used 0.85 as the threshold, based on our experiments.

Experimental results

We implemented A-C Pretenuring using ASH Tracker. Figure 10 shows the speedup results compared to the baseline that does not perform allocation site tracking or pretenuring. A-C Pretenuring improved sunflow by 15.7% and 11.8% with two times and four times the minimum Java heap sizes, respectively. On average, it delivered 3.8% and 1.8% speedups. Xml.transform and pmd were degraded by about 2%, due to the overhead of ASH Tracker and A-C Pretenuring. When the pretenuring was disabled and the allocation sites of every copied object were just looked up, the slowdown was within 2%. Thus ASH Tracker is an efficient infrastructure for allocation-site-based optimizations.

In sunflow, there were 10 allocation sites of `Color` class where pretenuring was turned on. That means there were 10 allo-

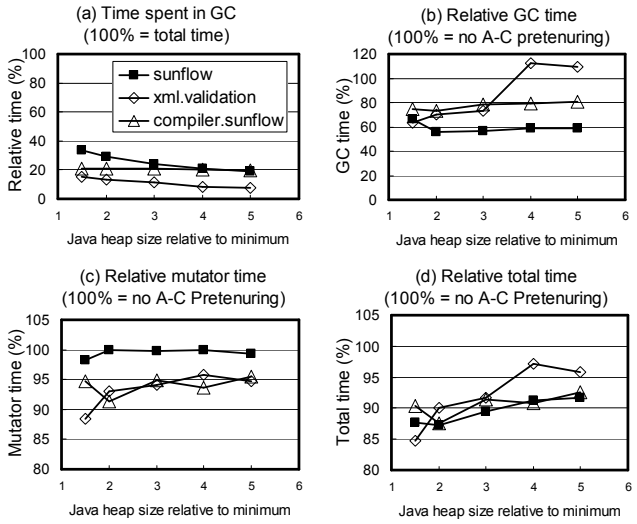

Figure 11. Statistics for GC and A-C Pretenuring in sunflow, xml.validation, and compiler.sunflow.

cation sites of the `Color` class whose tenuring ratio exceeded 0.85. We also found that there were 43 allocation sites of the `Color` class. The tenuring ratio of the `Color` class as a whole was 0.65, which indicates that pretenuring should be performed on a per allocation-site basis. Without pretenuring, 1.2% of the allocated objects were copied at least once in the young space (the last column of Table 1) and 70% of them were tenured. Note that although 1.2% was small, the reduction in GC time by pretenuring mostly depends on the ratio of the tenured objects (70%), which is quite high in sunflow, compared with other benchmarks. A-C Pretenuring successfully pretenured 89% of the tenured objects and erroneously pretenured 31% of the non-tenured objects.

Figure 11 shows statistics for selected three benchmarks, changing the Java heap size from 1.5x to 5x the minimum. In these experiments, we specified a fixed amount of workload rather than fixed time. Figure 11(a) is the fraction of time spent in GC when no pretenuring was performed. Figure 11(b) to (d) present relative GC, mutator, and total execution time, respectively, when A-C Pretenuring was enabled. The speedup in sunflow was because of the reduction in GC time, while the other two benchmarks also benefited from the reduced mutator time. These results of the mutators are similar to the ones previously reported [5] and considered to be due to increased memory locality.

6. Related Work

As described in Section 1, most existing approaches that can track allocation sites at runtime require space overhead proportional to the number of objects. Some of them append or prepend an additional field to all objects, while others record per-object information in a separate table indexed by an object identifier. The space overhead was reported [17] to be as much as 25% for a program that has many small objects. To make things worse, an additional field in an object reduces the effective size of the CPU caches, because it is typically in the same cache line as the other instance fields that contain actual data. The separate-table approach avoids such problems, but it does not work efficiently with a copying or generational GC because the object addresses cannot be used as object identifiers. The allocation site of an object can also be identified from its address if the objects allocated at the same site are arranged in a particular memory region [24]. This approach is not compatible with a copying or generational GC either. In contrast, the ASH and ASC Trackers can track the allocation sites of every object without any additional per-object space overhead. Thus they do not reduce the effective CPU cache size.

Bit-Encoding Leak Location, or Bell [7], can probabilistically track allocation sites by encoding an object address and its allocation site into a single bit in the object header. Bell performs decoding over all allocation sites and a subset of all objects, and reports the allocation sites that match significant numbers of objects. Since Bell is a statistical approach, it requires a sufficient number of samples to detect memory leaks with high confidence. In contrast, our memory leak detector makes it possible to detect a leak at any time during program execution and even to sense a symptom of a leak before leaking objects dominate the Java heap. Also, Bell is not useful for performance optimizations such as pretenuring because it cannot identify the allocation site for each object. Since Bell takes object addresses as input, it does not work well with those types of GC that move objects. Therefore, it is not suitable to use in pretenuring.

Previous runtime pretenuring approaches are based on object sampling. They associate allocation site information only with sampled objects, via weak references [1,15] or additional fields [23]. They typically sample objects at every *n*-byte allocation, where *n* is a tuning parameter. However, it is not clear how representative those sampled objects are, and they do not report on the relationship between the sampling rate and the accuracy of the pretenuring decision. In contrast, ASH and ASC Tracker allows us to detect pretenuring opportunities dynamically and accurately without additional per-object space overhead.

Bacon et al. [2] proposed several approaches to compress an object header. One of their techniques removes the hash code field. In that case, ASC Tracker should be used to track allocation sites. They also proposed stealing several bits from the class pointer field. ASC Tracker is compatible with such a technique by regarding the class pointer as an allocation site pointer.

7. Conclusion

This paper proposes novel techniques, the Allocation-Site-as-a-Hash-code (ASH) Tracker and the Allocation-Site-via-a-Class-pointer (ASC) Tracker, to track the allocation sites of every Java object at runtime. The ASH and ASC Trackers are the first techniques that enable various kinds of allocation-site-based online debugging and optimization in production environments. ASH Tracker encodes the allocation site ID of an object in a part of its hash code field. ASH Tracker does not steal any bits from the hash code field but instead regards the ID as part of it. ASC Tracker makes the class pointer field in an object header refer to the allocation site structure, which in turn points to the actual class structure. It is equipped with optimizations to mitigate the indirection overhead. Both the ASH and ASC Trackers are lightweight because they do not add any per-object fields. Our implementation of the ASH Trackers suffered from at most 1.4% and on average a 0% slowdown for all of the benchmarks we measured. ASC Tracker has a slightly larger slowdown of 2% at maximum and 1.0% on average, but it does not require a hash code field in the object headers.

This paper also demonstrates the effectiveness of minimal-overhead allocation site tracking for both the reliability and optimization. For reliability, we implemented an allocation-site-based memory leak detector. At any time during execution, our memory leak detector allows users to investigate the histories of the number of live objects on a per-allocation-site basis. The overhead was 1.1% on average, which makes it possible to use the memory leak detector in a production environment. For optimization, we

propose an Allocation-site-based Copy-time (A-C) Pretenuring. During young GC, it copies directly into a tenured space the objects allocated at certain allocation sites that have frequently allocated long-lived objects. Experimental results showed that A-C Pretenuring achieved up to a 11.8% speedup with 4x the minimum Java heap size. A-C Pretenuring is easy to implement by modifying only a few places in the GC. This is made possible for the first time by our trackers, because it depends upon inspecting the allocation sites of all of the copied objects at runtime.

Acknowledgments

We thank Vijay Sundaresan in IBM Toronto Lab and the members of the Systems group in IBM Research - Tokyo for their valuable suggestions. We are also grateful to anonymous reviewers for providing us with helpful comments.

References

[1] Agesen, O. and Garthwaite, A. Efficient object sampling via weak references. In *Proceedings of the 2nd International Symposium on Memory Management*, pp. 121-126, 2000.

[2] Bacon, D. F., Fink, S. J., and Grove, D. Space- and time-efficient implementation of the Java object model. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pp. 111-132, 2002.

[3] Bacon, D. F., Konuru, R., and Serrano, M. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pp. 258-268, 1998.

[4] Barrett, D. A., and Zorn, B. G. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 187-196, 1993.

[5] Blackburn, S. M., Hertz, M., McKinley, K. S., Moss, J. E. B., and Yang, T. Profile-based pretenuring. *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 1, pp. 1-57, 2007.

[6] Blackburn, S. M., Singhai, S., Hertz, M., McKinley, K. S., and Moss, J. E. B. Pretenuring for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 342-352, 2001.

[7] Bond, M. D. and Mckinley, K. S. Bell: bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 61-72, 2006.

[8] Chen, G., Kandemir, M., Vijaykrishnan, N., and Irwin, M. J. Field level analysis for heap space optimization in embedded java environments, In *Proceedings of the 4th International Symposium on Memory Management*, pp. 131-142, 2004.

[9] Cheng, P., Harper, R., and Lee, P. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pp. 162-173, 1998.

[10] Chilimbi, T. M. and Shaham, R. Cache-conscious coallocation of hot data streams. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pp. 252-262, 2006.

[11] Chilimbi, T. M. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pp. 191-202, 2001.

[12] DaCapo Benchmarks. http://dacapobench.org/

[13] Eclipse.org. http://www.eclipse.org/ .

[14] Grcevski, N., Kilstra, A., Stoodley, K., Stoodley, M., and Sundaresan, V. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pp. 151-162, 2004.

[15] Harris, T. L. Dynamic adaptive pre-tenuring. In *Proceedings of the 2nd international Symposium on Memory Management*, pp. 127-136, 2000,

[16] Hastings, R. and Joyce, B. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pp. 125-136, 1992.

[17] Hauswirth, M. and Chilimbi, T. M. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 156-164, 2004.

[18] Huang, W., Srisa-an, W., and Chang, J. M. Dynamic pretenuring schemes for generational garbage collection. In *Proceeding of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 133-140, 2004.

[19] IBM Java Diagnosis Guide 6. http://www.ibm.com /developerworks/java/jdk/diagnosis/ .

[20] Intel IA-32 Architecture Software Developer's Manual.

[21] Java SE 6 API Specification. http://java.sun.com/javase/6/docs/api/ .

[22] Java VM Specification. http://java.sun.com/docs/books/jvms/

[23] Jump, M., Blackburn, S. M., and Mckinley, K. S. Dynamic object sampling for pretenuring. In *Proceedings of the 4th International Symposium on Memory Management*, pp. 152-162, 2004.

[24] Novark, G., Berger, E. D., and Zorn, B. G. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation* , pp. 397-407, 2009.

[25] Novark, G., Berger, E. D., and Zorn, B. G. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation* , pp. 1-11, 2007.

[26] Novark, G., Berger, E. D., and Zorn, B. G. Plug: automatically tolerating memory leaks in C and C++ applications. *Technical Report* UM-CS-2008-009, University of Massachusetts, 2008.

[27] Qin, F., Lu, S., and Zhou, Y. SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs In *Proceedings of the 2005 International Symposium on High-Performance Computer Architecture*, pp. 291-302, 2005.

[28] Seidl, M. L. and Zorn, B. G. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 12-23, 1998.

[29] Seward, J. and Nethercote, N. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the 2005 Annual Conference on USENIX Annual Technical Conference*, pp. 17-30, 2005.

[30] Shaham, R., Kolodner, E. K., and Sagiv, M. Heap profiling for space-efficient java. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pp. 104-113, 2001.

[31] Standard Performance Evaluation Corporation. SPECjbb2005. http://www.spec.org/jbb2005/ .

[32] Standard Performance Evaluation Corporation. SPECjvm2008. http://www.spec.org/jvm2008/ .

[33] Sun Microsystems. HotSpot VM. http://java.sun.com/javase/ technologies/hotspot/index.jsp