Continuous Object Access Profiling and Optimizations to Overcome the Memory Wall and Bloat

Rei Odaira, Toshio Nakatani

IBM Research – Tokyo
1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan
{ odaira, nakatani } @jp.ibm.com

Abstract

Future microprocessors will have more serious memory wall problems since they will include more cores and threads in each chip. Similarly, future applications will have more serious memory bloat problems since they are more often written using object-oriented languages and reusable frameworks. To overcome such problems, the language runtime environments must accurately and efficiently profile how programs access objects.

We propose Barrier Profiler, a low-overhead object access profiler using a memory-protection-based approach called *pointer barrierization* and adaptive overhead reduction techniques. Unlike previous memory-protection-based techniques, pointer barrierization offers per-object protection by converting all of the pointers to a given object to corresponding barrier pointers that point to protected pages. Barrier Profiler achieves low overhead by not causing signals at object accesses that are unrelated to the needed profiles, based on profile feedback and a compiler analysis. Our experimental results showed Barrier Profiler provided sufficiently accurate profiles with 1.3% on average and at most 3.4% performance overhead for allocation-intensive benchmarks, while previous code-instrumentation-based techniques suffered from 9.2% on average and at most 12.6% overhead. The low overhead allows Barrier Profiler to be run continuously on production systems. Using Barrier Profiler, we implemented two new online optimizations to compress write-only character arrays and to adjust the initial sizes of mostly non-accessed arrays. They resulted in speed-ups of up to 8.6% and 36%, respectively.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – optimization

General Terms Algorithms, Performance, Experimentation.

Keywords Memory protection; memory management; profiling

1. Introduction

Increasing numbers of cores and threads on each chip have reduced the per-core and per-thread caches and memory bandwidth. This trend results in the problem called *memory wall*, a long latency for programs to access data in memory. Typical programs written in object-oriented languages such as Java exacerbate the memory wall because they allocate and access a large number of objects [31]. Such programs are called *allocation-intensive* programs. Allocating many objects also causes frequent GC invocations and increases the GC overhead.

Recent research [14,29] has shown that a large fraction of those object allocations are wasted, since they do not involve any data that actually contribute to the output of the program. This problem is often called *memory bloat*. Memory bloat happens because more programs are written using reusable libraries and frameworks. Such libraries and frameworks can allocate large data structures for generic usage but are not optimal for particular use cases. Memory bloat not only interferes with the efficient use of memory, but also makes the memory wall even higher since more of the actually useful data is evicted from CPU caches.

To reduce the effects of the memory wall and memory bloat, language execution environments such as Java VMs need to profile the program accesses of objects and then to optimize those objects. Object access profiles include properties such as write-only objects, immutable objects, and non-accessed bytes based on the information about which program instructions access which fields of which objects. Object access profiling is crucial for many object optimizations, including object compression [5,12,21], lazy allocation [5,24], field reordering [6], and object merging [12]. For example, if certain objects are unlikely to be accessed after initialization, they should be compressed.

There are two requirements for an object access profiler. First, it must be lightweight so that it can be used online. Although the object optimizations can be done offline by hand or by a compiler, offline optimizations cannot capture the dynamic behavior of programs. Thus language execution environments should profile and optimize object accesses online. Second, it must be accurate for maximum performance, especially when using speculative object optimizations. Because many object optimizations are speculative, inaccurate profiles can result in speculation failures. To obtain accurate object access profiles, the profiler must track the entire lifetimes of objects, not small portions. For example, to determine immutability the profiler must confirm that all reads come after all writes to an object. Thus it must be able to detect all of the accesses to the object, or at least all of the accesses that affect the accuracy of the access profiles.

In this paper, we propose *Barrier Profiler*, an accurate object access profiler with low performance overhead. It is based on per-object memory protection combined with profile-directed adaptive techniques for overhead reduction. Barrier Profiler uses *pointer barrierization*, which reserves a protected region outside of the heap region, instead of protecting the pages containing the profiled objects. At object allocation time, pointer barrierization converts all of the pointers to a given object to corresponding

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'12, March 3–7, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-0759-8/12/03…$10.00.

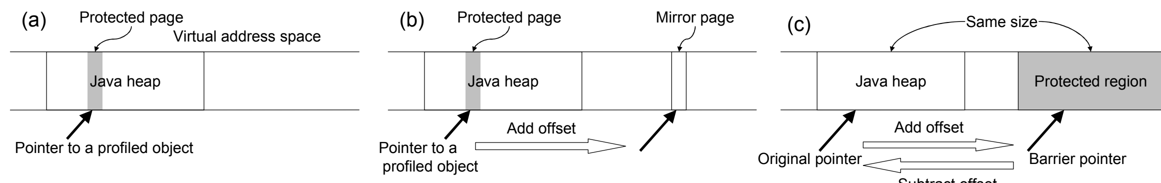

Figure 1. (a) Basic memory protection causes a race condition. (b) Mirror pages still suffer from coarse-grain page-level protection. (c) Pointer barrierization provides per-object protection.

barrier pointers that point to the protected region. All of the accesses via the barrier pointers cause hardware exceptions, allowing the profiler in the exception handler to track the lifetime of the object. Since hardware exceptions occur only via the barrier pointers, the protection granularity is not per page but per object.

Using pointer barrierization as a substrate, Barrier Profiler profiles frequently-allocated short-lived objects, because such objects affect the performance by polluting CPU caches and squandering memory bandwidth, especially in multi-core systems [31]. To reduce the profiling overhead, Barrier Profiler samples only a small number of objects at each allocation site and profiles the accesses to the sampled objects. Barrier Profiler further reduces the overhead by profile-directed adaptive object sampling and *unbarrierization*. If the target of an object access optimization is an immutable object, for example, what is crucial is not the accurate numbers of accesses but the object accesses that affect the immutability. Thus Barrier Profiler can skip sampling the objects that are unlikely to be immutable and profiling the objects that are no longer immutable. Our experiments showed that these techniques provided sufficiently accurate profiles with low overhead. Thus Barrier Profiler can run continuously in production systems.

We implemented two new online object-access optimizations to demonstrate the usefulness of Barrier Profiler. One is an online speculative compression of character arrays. Based on the online feedback from Barrier Profiler, the arrays that are unlikely to be accessed after their initialization are compressed if they contain only ISO-8859 characters. The other optimization is dynamic adjustment of the initial sizes of container arrays. Programmers often over-allocate large arrays as buffers, but access only the first few elements. Barrier Profiler successfully identified optimal sizes based on the non-accessed-byte profiles and dynamically performed recompilation to inline the allocation contexts and to embed the optimal sizes as compile-time constants.

In summary, our contributions are:

- We propose Barrier Profiler, an accurate object access profiler with low performance overhead, which is suitable for continuous profiling in the field. Per-object accurate profiling is enabled by pointer barrierization (Section 3). Barrier Profiler is lightweight because of its profile-directed adaptive object sampling and unbarrierization (Section 4).
- We implemented Barrier Profiler in the IBM J9/TR [7], a production-quality Java VM with a just-in-time (JIT) compiler. We conducted experiments on its accuracy and overhead (Section 5), using the industry standard SPECjvm2008, SPECjbb2005, and DaCapo benchmarks.
- We quantitatively compared Barrier Profiler with existing code-instrumentation-based techniques including Bursty Tracing [1,8] (Section 5).
- We implemented and evaluated two new online object-access optimizations to demonstrate the usefulness of Barrier Profiler (Section 6). Both of them are feasible for the first time by using the lightweight object access profiler.

2. Related Work

Code instrumentation [2,4,5,12,24,30] has been a standard technique for object access profiling. However, executing additional code at every heap access results in prohibitive overhead. Both memory and performance overheads were reportedly increased by more than a factor of 10 [12]. One way to mitigate the overhead is access sampling like Bursty Tracing [1,8], which enables the instrumentation only during a profiling phase. However, our experiments showed excessive overhead even at low sampling frequencies.

In addition, the access sampling approaches are inaccurate because they can only sample some fraction of the accesses to arbitrary objects. They cannot track the life of an object, for example to detect immutable fields [26] or to find last access points [22]. What is needed is a lightweight profiler that samples objects and detects *all* of the accesses to the sampled objects, or at least all of the accesses that affect the accuracy of the access profiles.

To detect the accesses to the objects without the code instrumentation overhead, memory protection with signal handling has been used in many systems [15,16,19,25,27,30]. Hound [16] and pointer swizzling [27] simply protect the pages that contain the target objects, as shown in Figure 1(a). However, this basic protection does not work for object access profiling because of race conditions. For example, Hound uses the basic protection to confirm the staleness of the objects that are segregated at allocation time. Once the application accesses a protected page, all of the objects in that page become fresh and the page is unprotected. In contrast, for more general object access profiles such as write-only objects, immutable objects, and non-accessed bytes, the page cannot be unprotected after just one access to an object being profiled. However, when a thread accesses a protected object, it must temporarily disable the protection to access its content. In the meantime, race conditions can occur because other threads might access that object or other objects contained in the same page without causing hardware exceptions.

To avoid the race conditions, Memalyze [25], Archipelago [15], and other systems [19,30] use mirror pages that map the physical pages of the protected virtual pages to other virtual addresses, as shown in Figure 1(b), allowing the signal handler to access the data of the protected object without unprotecting the page that contains it. However, these techniques still suffer from too much performance overhead to use online. First, the signal handling by an OS can result in more than a 50% performance overhead in allocation-intensive programs [16]. It is important to reduce the number of hardware exceptions without compromising the profile accuracy. Second, the page-level protection is too coarse-grained. Accesses to objects that are not being profiled but that reside in the protected page suffer from hardware exceptions. The granularity problem can be mitigated by segregating the profiled objects into the protected pages [16,30]. However, this requires a new memory allocator specialized for those protected pages, complicating the system design.

Pointer barrierization enables per-object protection by enhancing the mirror page approach. It does not protect the pages containing the profiled objects but instead reserves a protected region outside of the heap region, as shown in Figure 1(c). Chen et al. [5] presented a very basic form of pointer barrierization to detect accesses to compressed objects. After compressing objects during GC, their approach sets the highest order bit of each pointer to 1 for each compressed object, based on the assumption that an embedded system will not use a virtual address space larger than 2GB. Their original technique uses code instrumentation, and as an enhancement, they described a technique to use hardware memory protection. However, they did not explain the details of their algorithm. Li et al. [11] used a similar technique to detect accesses to aliased stack slots. In fact, pointer barrierization will not work correctly on objects in the heap if internal pointers and atomic instructions are ignored. These details are addressed for the first time in our work. Also, we show that pointer barrierization is useful for object access profiling when combined with the overhead reduction techniques.

QVM [2] detects defects in production systems using techniques similar to Barrier Profiler. It continuously monitors accesses and method invocations on the objects that are sampled adaptively. However, because it uses code instrumentation, users of QVM need to specify which program points to instrument to avoid the large overhead of instrumenting every point. Also, its adaptive object sampling is based only on system overhead, while ours is profile-directed and thus is more suitable for insuring the accuracy of the collected object access profiles.

There have been many other research proposals to overcome memory bloat. Xu et al. proposed a static analysis [28] and a profiling method [29] to reduce memory bloat in Java. However, they were designed for offline code rewriting. The runtime slowdown by the profiling was more than 10-fold. Sartor et al. [21] estimated potential space savings by various existing object compression algorithms, but they did not focus on how to implement them online. In contrast, Barrier Profiler is a novel infrastructure for online reduction of memory bloat. Discontiguous arrays with lazy allocation [5,20] can dynamically reduce memory bloat in large arrays. However, even when using highly optimized z-rays [20], performance degradation was more than 10% on average. Our dynamic adjustment of initial array sizes using Barrier Profiler has the same effect as using discontiguous arrays for certain array usage patterns, but has much lower overhead than z-rays. Chameleon [23] can also find similar opportunities through code instrumentation, but its performance overhead is 6 times at maximum.

3. Pointer Barrierization

In this section, we describe a pointer barrierization algorithm and what needs to be considered in code generation and GC. Pointer barrierization enables per-object access detection and is compatible with all types of memory allocator. Throughout this paper, we assume Java is our target environment, but pointer barrierization and Barrier Profiler could be adapted to the environments of other languages and be effective on both 32-bit and 64-bit architectures.

3.1 Basic pointer barrierization

The easiest pointer barrierization is to add to a pointer the offset between the Java heap region and a read-write-protected region whose size matches the heap region. Figure 1(c) shows a memory layout. Note that the protected region does not require the assignment of real memory.

Pointer barrierization can be performed at allocation or GC time. For pointer barrierization to work correctly, it is important to barrierize all of the pointers that point to a target object. Since only one pointer in a register points to an allocated object immediately after allocation, it suffices to barrierize that register in the generated JIT code or in a modified interpreter. GC can do pointer barrierization when traversing all of the pointers to the object.

3.2 Virtual-memory-efficient pointer barrierization

The basic pointer barrierization requires a protected region whose size is not smaller than the Java heap region. In a 32-bit environment, however, the virtual address space is a limited resource. It is best to make the protected region as small as possible.

For a smaller protected region, Java VM can take advantage of the fact that all Java objects are aligned on a 4- or 8-byte boundary. This is true for production-quality Java VMs [7][17]. Since the last 2 or 3 bits of each object pointer are zero, it can be shifted right by 2 or 3 bits without losing any information. Thus the size of the protected region can be 1/4 or 1/8 that of the Java heap region. A difficulty in this approach is that an access through a barrier pointer can point outside of the protected region. Thus when barrierizing the pointers to an object, the sum of the barrier pointer and the object size must not exceed the top address of the protected region. If this condition fails, barrierization cannot be used, but this is a rare case.

In some CPU architectures, unaligned accesses via a pointer whose last bits are set cause a bus error. However, pointer barrierization cannot take advantage of this mechanism in general, because byte accesses through such a pointer do not cause the error.

To further reduce the size of the protected region, a Java VM can use an OS-protected region. For example, by default the Linux and Windows OS [13] occupies the highest 1/4 or 1/2 of a virtual address space, and cannot be accessed by users. In this case, pointer barrierization can shift a pointer right by at least 2 bits and set the topmost 2 bits. The Java VM does not need to reserve any protected region in the user space.

3.3 Signal handling

Accesses through barrier pointers always cause hardware memory exceptions, which are converted to signals by the OS. Figure 2 shows the signal-handling algorithm implemented in a Java VM.

Restoring a header pointer

The handler first checks whether the excepting data address is within the protected region by address comparison (Line 2). If so, the handler must restore the pointer to the head of the accessed object, which we call a *header pointer*. A pointer to the middle of the object, which we call an *internal pointer*, is not sufficient because an object access profiler must identify which object is being accessed to detect, for example, the immutability of the object (Line 11). Generally, unbarrierizing the excepting data address only produces an internal pointer.

We propose two approaches to restore the header pointer. One is to use an ordered set, normally a balanced tree. At pointer barrierization time, the barrier pointer, which corresponds to a header pointer to an object, is stored in the ordered set. To restore the header pointer, the highest barrier pointer that is lower than the excepting data address is looked up in the ordered set (Line 5). This search requires synchronized tree traversal and can result in a bottleneck in multithreaded execution.

The other approach is to restrict the format of the instructions that access objects. This means one of the register arguments of a memory reference must be a header pointer. For example, in a memory reference of the form `[base_reg, offset_reg/immediate]`, `base_reg` must be a header pointer.

```
01: handle_signal(code_addr, data_addr, context) {
02:   if (is_in_protected_region(data_addr)) {
03:     base_reg_num = decode_instruction(code_addr);
04:     base_reg = get_reg_contents(base_reg_num, context);
    #if USE_ORDERED_SET
05:     bar_object_head = search_ordered_set(data_addr);
    #else
06:     if (is_internal_pointer_access(code_addr))
07:       bar_object_head = get_array_head(code_addr);
08:     else
09:       bar_object_head = base_reg;
    #endif
10:     orig_object_head = unbarrierize(bar_object_head);
11:     record_access(code_addr, orig_object_head);
12:     if (is_LL(code_addr) ||
13:         is_uninteresting(orig_object_head) &&
14:           safe_to_unbarrierize_at(code_addr)) {
15:       write_to_context(context, base_reg_num,
            base_reg + orig_object_head - bar_object_head);
16:       return_to(code_addr);
17:     }
18:     do_load_or_store(code_addr, orig_object_head, context);
19:     return_to_next_instruction(code_addr);
20:   }
21:   ...
```

Figure 2. Signal handling algorithm.

Then the barrierized header pointer can be obtained by decoding the instruction and loading the content of `base_reg` from the signal context (Lines 3-4). We believe that these restrictions are usually satisfied in existing Java VMs and JIT compilers.

However, the restrictions may be violated when accessing large objects, especially arrays, due to the results of loop induction variable optimization. For those instructions that use internal pointers, JIT-generated code maintains a header pointer in a register or in a stack slot of the method. Many JIT compilers already support this function to help the GC mark or move the objects pointed to by internal pointers. In addition, the JIT compiler maintains a hash table to associate each instruction that uses an internal pointer with a register number or a stack offset that holds the header pointer. This allows the signal handler to obtain the header pointer from the excepting code address (Lines 6-7). Since most of the object access instructions do not use internal pointers, the data structure for the association will not grow excessively.

The second approach is more lightweight in general, but we used the first approach because our preliminary experiments showed that its overhead was small enough, thanks to the low object sampling frequency of our Barrier Profiler.

Handling atomic instructions

After unbarrierizing the header pointer (Line 10), the handler records the access (Line 11) and emulates the execution of the excepting instruction (Line 18). For example, emulating a load means executing a load on behalf of the excepting load instruction and then writing the loaded value to the target register in the signal context. Note that the handler is not allowed in general to write the unbarrierized pointer to the base register and return to the excepting instruction to execute it again. This approach could change the semantics of a program because the same object is represented by two different pointer values, causing a pointer equality operation to return false even when the two pointers do refer to the same object. After the emulation, the handler returns to the next instruction (Line 19).

There is a subtle problem in handling atomic instructions (Lines 12, 15-16). The signal handler can correctly emulate a compare-and-swap (CAS) instruction. However, for a load-linked (LL) instruction, even if the handler executes an LL, that reservation is likely be lost before the execution reaches a corresponding store-conditional (SC) instruction. This is because the OS can execute another LL/SC pair after the signal handler and before the SC. We solve this problem by executing the LL again with an unbarrierized register, as shown in Figure 3. To avoid the problem of incorrect pointer equality, the base register is copied to another register immediately before the LL, and the LL uses that register. Thus the original base register still contains a barrier pointer. In addition, the following SC must also use the copied register. Otherwise, another hardware exception would occur at the SC, and the reservation might be lost. In Java, LL/SC pairs can be performed on objects only for monitor entrances and exits, so it is easy to modify a JIT compiler or Java VM to use LL/SC in the required way. The conditions in Lines 13 and 14 reduce the overhead and are explained in Section 4.2.

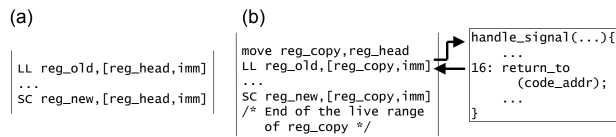

Figure 3. (a) Original code sequence for LL/SC in JIT or interpreter code. (b) With barrier pointers, a base register is copied to another register (`reg_copy`), and LL/SC use that register. If the LL causes an exception, the signal handler modifies `reg_copy` to an unbarrierized pointer, and the LL is executed again.

3.4 GC

With pointer barrierization, most GC implementations need only minor changes to run correctly. An efficient marking GC uses a marking table outside of the Java heap region. It finds the corresponding mark bit of an object by bit-operations on the pointer to the object. Thus, before marking, the GC must check whether or not the pointer is a barrier pointer. If so, it must obtain the original pointer by unbarrierization. The same operation is needed for marking a card table for generational or concurrent GC.

In the GC algorithms that move objects, each pointer field is modified so that it points to the new location of the pointed-to object. If the original pointer is a barrier pointer, the new pointer must be one, too. Thus GC needs a check at each pointer traversal, and it must perform unbarrierization and barrierization as needed.

4. Barrier Profiler

Barrier Profiler is a lightweight object-access profiler using pointer barrierization. It is focused on frequently allocated short-lived objects, because such objects often affect the performance in multi-core systems [31]. Pointer barrierization could also be used for profiling long-lived objects. This is left for future work.

4.1 Basic algorithm

To profile the frequently allocated objects, the interpreter and JIT compiler were modified to use pointer barrierization at all allocation sites. To reduce the profiling overhead, Barrier Profiler applies pointer barrierization to only a small fraction of the objects at each allocation site. This object sampling should be based not on the allocated number of objects but on the allocated bytes, because cache pressure, memory bandwidth usage, GC frequency, and memory bloat are all related to the allocated bytes.

Figure 4 shows the pseudocode at an allocation site. After allocating an object (Line 1), the addresses of the head and tail of the object are XORed and ANDed with a mask to check whether the object is allocated across an aligned boundary (Line 3). The per-allocation-site mask is of the form 11…100…0. For example,

```
1: object = allocate_object(class);
2: object_tail = object + object_size;
3: if (((object ^ object_tail) & mask[site_id]) != 0) {
4:   record_allocation(site_id, object);
5:   object = barrierize(object);
6:   mask[site_id] = get_next_mask(site_id);
7: }
```

Figure 4. Code at an allocation site in Barrier Profiler.

if the number of zeros is 20, Barrier Profiler samples the objects that extend across 1-MB-aligned boundaries at this allocation site. More zeros in the mask mean lower overhead but larger errors in the profiles. If the masked results are not all zero, then the allocation is recorded (Line 4) and the pointer is barrierized (Line 5). The mask can be updated as needed (Line 6).

Barrier Profiler records the per-object access profiles in object information structures. One object information structure corresponds to one sampled object. It is created outside of the Java heap at the allocation time of the sampled object (Line 4) and contains the ID of the allocation site. In addition, the allocation context is obtained by stack walking, and the ID assigned to the context is stored in the information structure. When a sampled object is accessed (Line 11 of Figure 2), the corresponding information structure is fetched via a hash table using the object head address as a key. Alternatively, if the ordered set is used to restore a header pointer, the set can also be used as a map to store and to fetch the information structures in addition to header pointers. Per-object access profiles such as the number of reads and writes, whether the object is write-only or not, etc. are recorded in the object information structure. At the end of each GC, every object information structure is visited to check whether or not its corresponding object is dead. If it is dead, the per-object access profiles are accumulated into the per-allocation-site and per-allocation-context information structures that correspond to the allocation site and context, respectively, of the dead object. The object information structure is then reclaimed.

4.2 Performance overhead reduction

We devised six techniques to reduce the performance overhead of Barrier Profiler by decreasing the number of hardware exceptions: three that lower the frequency of object sampling and three that unbarrierize some of the objects being profiled. A key constraint is to avoid compromising the accuracy of the profiles.

Adaptive object sampling

There are three cases where the sampling frequency can be decreased without increasing the profile errors too much. First, Barrier Profiler can reduce the sampling frequency at uninteresting allocation sites based on the object access optimizations. For example, if a target optimization is to lazily allocate non-accessed objects, the objects allocated at a site that has allocated some or many non-accessed objects should be sampled frequently to carefully assess whether or not it is worth delaying object creation at this site. In contrast, the objects allocated at a site that has always allocated accessed objects can be sampled less frequently, although the sampling should not be completely disabled to adapt to dynamic behavior changes. Based on the feedback from the per-allocation-site profiles, our implementation lowers the sampling frequency at the allocation sites whose ratios in bytes of write-only objects, immutable objects, and non-accessed bytes to all of the objects sampled at the sites are all less than 1%.

Second, we found a few small objects received quite a large number of accesses in some benchmark programs. Such a small number of small objects have little effect on the memory wall or bloat problems. If they happen to be sampled, however, performance degradation is significant. Based on the per-allocation-site profile, Barrier Profiler decreases the sampling frequency at the allocation sites (1) whose sampled objects account for only a small portion (less than 0.5% in bytes, in our implementation) of all sampled objects, and (2) that have ever allocated an object whose ratio of the number of accesses to its size exceeded a threshold (50, in our implementation).

Finally, Barrier Profiler should sample large objects less frequently than small ones because the algorithm in Figure 4 can sample too many large objects, especially in scientific applications. Preliminary experiments show that we should treat objects from 1 KB and up as large objects.

Adaptive unbarrierization

Barrier Profiler stops profiling particular objects in three cases: two at GC time and one during the execution of the program. To disable the profiling of an object, all of the barrier pointers to the object must be unbarrierized. Otherwise, partial unbarrierization could cause incorrect pointer equality as described in Section 3.3.

First, if an object being profiled no longer satisfies the property of the profiles being collected, such as immutability, profiling of accesses to the object can be stopped. After Barrier Profiler records an access to an object, it checks whether the object is still worth profiling. If not, it sets a flag in the corresponding object information structure, indicating the object is no longer interesting. Barrier Profiler performs the unbarrierization of the uninteresting object during stop-the-world GC, when all of the pointers to the object are traversed.

Second, Barrier Profiler also does temporary unbarrierization during the execution of the program, so that the uninteresting object does not continue to be profiled until the next GC. In the signal handler, a barrier pointer in the base register can be unbarrierized and written back, as shown in Line 15 of Figure 2. As a result, the following accesses using this base register do not cause hardware exceptions. To avoid the problem of incorrect pointer equality, a compiler analysis insures that this base register will not be used as an operand of any pointer comparison instructions. Because our implementation does not use alias or inter-method analysis, the base register also must not be used as an argument of method invocations, a method return value, or for data to be stored in the heap. If the base register satisfies these conditions, the address of the instruction is registered in a global hash table. In the signal handler, if the object is uninteresting and the excepting address is registered in the table (Lines 13-14 of Figure 2), the base register is unbarrierized.

Finally, many long-lived objects receive a large number of accesses without changing their object access properties such as immutability. Thus Barrier Profiler stops profiling long-lived objects based on how many times they survive GC. Each time an object survives GC, its age is incremented in its object information structure. After its age exceeds a predefined threshold (8, in our implementation), it is unbarrierized at the next GC.

Discussion

Some of the overhead reduction techniques cannot be used when collecting certain types of object access profiles. For example, profiling the last access points of objects is useful for inserting cache-flush instructions. However, there is no known time when an object becomes “uninteresting” in terms of its last access point. Thus sampled objects should not be unbarrierized.

5. Experimental Results

This section describes our implementation of Barrier Profiler. Then our experimental results are presented to assess the accuracy and overhead of Barrier Profiler.

5.1 Implementation

We implemented Barrier Profiler in a development version of the 32-bit IBM J9/TR [7] 1.6.0 SR6 for the Power architecture [18]. We used the basic pointer barrierization described in Section 3.1. We allocated 1 GB for the Java heap and also reserved 1 GB for the protected region. We used mark-and-sweep GC that invoked occasional compaction when the fragmentation became excessive. We measured the effects of the six overhead reduction techniques described in Section 4.2 by enabling and disabling all of them, which we call Opt and NoOpt, respectively.

We modified the interpreter to execute the pointer barrierization code in Figure 4 and the JIT compiler to insert the code into all of the allocation sites in the JIT-compiled methods. J9/TR first executes Java methods with the interpreter, and if a method is frequently invoked, then it is JIT-compiled. We inserted the pointer barrierization code immediately after the initialization of the object header fields and the zero-clearing of the non-header fields. This means these initialization-related writes did not appear in the collected profiles.

For the mask value in Figure 4, we used a fixed value during each run of a benchmark, except for the optimizations in the Opt configuration. We tried five mask values, with 17, 20, 23, 26, and 29 zeros. This corresponds to sampling one object per each allocated 128 KB up to each 512 MB. In this section, we use these numbers of sampling intervals to denote each configuration. In the JIT-compiled code, the mask is embedded as an immediate value in the AND instruction at Line 3 of Figure 4. The mask value can be updated by code patching. In the Opt configuration, we increased the number of masking zeros by three when decreasing the sampling frequency. For object allocation in the interpreter, our implementation uses a global fixed mask value.

The total real memory requirement of Barrier Profile was less than 8 MB even at the most frequent sampling interval of 128 KB. More than 90% of the memory was for the information structures described in Section 4.1. Note that these data structures are not Barrier-Profiler-specific. They are needed for code-instrumentation-based object access profilers too.

5.2 Code instrumentation techniques

We implemented two code instrumentation techniques in J9/TR for comparison. The first one is full instrumentation to simulate Bursty Tracing [8]. Bursty Tracing is based on the Arnold-Ryder framework [1] and aims for temporal profiling similar to Barrier Profiler. It duplicates the JIT-compiled code for every method. One is a checking version and the other is an instrumented version. Most of the time, the checking version is executed, but the execution occasionally switches to the instrumented version. The checking version contains only checks at method entries and loop back-edges that decrement a counter and jump to the instrumented version if it reaches zero. The instrumented version also contains the same checking code for the other counter to jump back to the checking version. Although we did not implement all of the parts of Bursty Tracing, we can estimate its overhead by the sampling rate and by the overhead of the instrumented code, as explained in [8]. To that end, we implemented full instrumentation that inserts profiling code at every heap access.

We also implemented another code instrumentation technique based on the idea of Barrier Profiler. We call it *Instrumented Barrier Profiler*. It performs pointer barrierization at allocation sites as in Barrier Profiler, but instead of relying on hardware exceptions, it inserts check instructions at every heap access. It uses the basic pointer barrierization in Section 3.1, although we did not need to actually reserve the protected region. To make the check simple, we guaranteed that the lower 16 bits of the bottom address of the protected region were all zero. The check consists of three instructions: The first instruction logically shifts a pointer right by 16 bits. The second instruction uses a 16-bit immediate comparison instruction to compare the shifted value with the constant that is the higher 16 bits of the bottom address of the protected region. The third instruction is a branch instruction based on the result of the comparison. If the pointer is equal to or larger than the bottom address, the execution jumps to a profiling snippet, which is basically the same as the signal handler in Figure 2. We also implemented a compiler optimization to merge instrumentations for the same object within a basic block, combined with temporary unbarrierization in the snippet. Instrumented Barrier Profiler can also benefit from the optimizations in Section 4.2. To obtain the same profiling accuracy as Barrier Profiler, Instrumented Barrier Profiler must always be enabled.

5.3 Benchmarks and evaluation environment

To evaluate the accuracy and overhead of Barrier Profiler, we used SPECjvm2008, the DaCapo Benchmarks 9.12, and SPECjbb2005. We ran the benchmarks six times, restarting the JVM each time, and took the averages during the steady state. We ran the benchmarks on a 4-core 2-way-SMT 4.0-GHz POWER6 [10] machine with 32 GB of main memory running Linux 2.6.18. For SPECjvm2008, we ran a 2-minute warm-up and a 4-minute measurement iteration. We used small data sets for the "scimark" benchmarks in SPECjvm2008. For DaCapo, we used the largest data sets for each benchmark. We ran each benchmark for 6 minutes and used the last 4 minutes as a measurement period. We excluded eclipse, tradebeans, and tradesoap from DaCapo, because they did not run correctly on the development version of the JVM we used, even without Barrier Profiler. For SPECjbb2005, we used a single JVM, which ran a 3.5-minute warm-up and a 4-minute measurement run with 8 warehouses.

The main targets of Barrier Profiler are allocation-intensive programs that allocate and use large numbers of objects, causing memory wall and memory bloat problems. Zhao et al. [31] categorized the SPECjvm2008 programs into allocation-intensive and non-allocation-intensive programs and pointed out that the allocation-intensive ones suffered from serious scalability issues on multi-core systems. Table 1 shows the per-hardware-thread allocation rates of the benchmarks we used. The rates are normalized to scimark.montecarlo as 1. The left side is allocation-intensive, while the right side is non-allocation-intensive. In SPECjvm2008, compiler.compiler, derby, serial, sunflow, xml.transform, xml.validation, and compiler.sunflow are allocation-intensive. These results match Zhao et al.'s categorization.

5.4 Collected object access profiles

Using Barrier Profiler, we collected three kinds of object access profiles: write-only objects, immutable objects, and non-accessed bytes. In our experiments, we collected all of the three profiles at the same time. Write-only objects are those objects for which none of the non-header fields are read. These objects are useless for the executed paths of the benchmarks, making them candidates for object optimizations such as allocation-time compres-

sion or lazy allocation [5,24]. Barrier Profiler can estimate how frequently each allocation site allocates write-only objects. If an allocation site always or mostly allocates write-only objects, then that site should be a target for the object optimizations.

The second object access profile we collected is for immutable objects. These are the objects for which all of the writes to their non-header fields precede all of the reads from their non-header fields. They can be considered as runtime constant objects, and thus they can be exploited for lazy allocation, or more specifically, for copy-on-write. Note that this kind of object access profiling cannot be collected by using access-sampling-based methods, because such methods cannot track the entire life of a particular object. In our definition, all of the write-only objects are also immutable objects.

The last object access profile is the non-accessed-byte profile. A non-accessed byte in an object is a non-header byte for which there is no read or write after the header initialization and zero-filling. Such a byte only wastes space and should be eliminated. Note that this is a per-byte property, while the previous two profiles are per-object properties.

5.5 Accuracy of profiling

To evaluate the accuracy of object access profiling, we need to have perfect reference profiles, which we also obtained with a modified form of Barrier Profiler. We added to Barrier Profiler a mechanism to enable or disable the profiling. Pointer barrierization code at allocation sites was skipped by branch instructions during start-up. After the program reached a steady state, the profiling was enabled by patching the branch instructions to no-ops. This profiling was different from that in Figure 4 in that it sampled *all* of the allocated objects. After the patching, the program continued execution to the end, profiling all of the accesses to the objects. We call these results the “full trace”. For evaluation, we also counted how many bytes of the objects were actually allocated at each allocation site in the steady state. We call this the actual allocated bytes of the allocation site.

Here is how we evaluated the accuracy of object access profiles such as for the write-only objects. For each allocation site, we computed an estimated write-only ratio by dividing the total bytes of the sampled write-only objects allocated at the site by the total bytes of all of the sampled objects allocated at the site. If no object was sampled at an allocation site, we assumed the estimated write-only ratio of the site to be zero. We also calculated an actual write-only ratio for each allocation site, using the full trace results. The estimation error in the allocated bytes of write-only objects is the total of the actual allocated bytes of each allocation site multiplied by the absolute difference between its estimated and actual write-only ratios. In other words, the estimation error is the average of the absolute differences weighted by the actual allocated bytes. We computed estimation errors for immutable objects and non-accessed bytes in the same way.

Since Barrier Profiler is sampling-based, long-term profiling will eventually reach the correct values. However, it should return the correct values even from short-term profiling, so that it can react quickly to dynamic behavior changes. Thus for the experiments in Section 5.5 and 5.7, we report the profiling results obtained during the measurement period of each benchmark, as described in Section 5.3. The profiling is always enabled, but all of the profiles collected during the warm-up period are discarded before the measurement period. Investigating the sensitivity of the accuracy to the profiling length is left for future work.

Figure 5 shows the estimation errors for write-only objects, relative to the total actual allocated bytes in compiler.compiler, derby, xml.transform, and xml.validation. Here we show only the results of the 4 representative allocation-intensive benchmarks, and we omit the results for immutable objects and non-accessed bytes. The estimation error ratios mostly decreased as the sampling interval was reduced except in derby, where the error ratios were always less than 2%. At 8-MB/Opt, the estimation error ratios in the allocation-intensive benchmarks were within 10%, 16%, and 6% for the write-only objects, immutable objects, and non-accessed bytes, respectively. The estimation error ratios of immutable objects were larger than those of write-only objects and non-accessed bytes, because the immutable objects accounted for more than 50% of the total actual allocated bytes in most benchmarks, while the write-only objects and non-accessed bytes rarely exceeded 20%.

Allocation-intensive benchmarks	Norm. allocation rate	Non-allocation-intensive benchmarks	Norm. allocation rate
compiler.compiler	15.8	compress	1.3
derby	39.3	crypto.aes	11.7
serial	20.5	crypto.rsa	1.3
sunflow	16.4	crypto.signverify	3.7
xml.transform	21.5	mpegaudio	2.1
xml.validation	24.8	scimark.fft	4.2
compiler.sunflow	15.4	scimark.lu	6.4
fop	33.2	scimark.sor	6.5
jython	40.7	scimark.sparse	1.0
lusearch	850.3	scimark.montecarlo	1.0
pmd	27.8	avrora	1.3
sunflow (DaCapo)	119.8	batik	5.0
tomcat	71.6	h2	6.1
xalan	128.1	luindex	3.6
SPECjbb2005	16.8		

Table 1. Normalized per-hardware-thread allocation rates of the benchmarks (bytes/second). The rates are normalized to scimark.motecarlo as 1. The targets of Barrier Profiler are the allocation-intensive benchmarks on the left side.

When comparing Opt with NoOpt, the estimation error ratios were almost always worse, but within 5% difference. Further investigation revealed that most of the inaccuracy in Opt resulted from lowered sampling frequencies at uninteresting allocation sites. Surprisingly, Opt sometimes resulted in smaller estimation error ratios than NoOpt, as shown in compiler.compiler and xml.validation of Figure 5. This was because NoOpt was so slow that it could not sample enough number of objects during the measurement period.

5.6 Performance overhead of profiling

How accurate object access profiling should be depends on how much overhead the profiling incurs and how sensitive an object access optimization is to the errors in the profiles. Figure 6 presents the relative performance overheads of all of the Barrier Profiler configurations in the 4 representative allocation-intensive benchmarks, compared with the baseline without Barrier Profiler. The sampling intervals of 1 MB or less suffered from significant overhead. For example, in compiler.compiler, 512MB/Opt caused $1.5 * 10^4$ profiled accesses per minute, 8MB/Opt did $1.5 * 10^6$, and 128KB/Opt did $2.2 * 10^7$. Based on these results, we chose 8-MB as the best to balance accuracy with overhead.

When comparing Opt with NoOpt, their differences were statistically significant at the 8-MB-or-shorter intervals in most benchmarks. At the 8-MB interval, the adaptive object sampling contributed to most of the overhead reduction. The GC-time unbarrierization rarely occurred at the 8MB-or-longer intervals. As

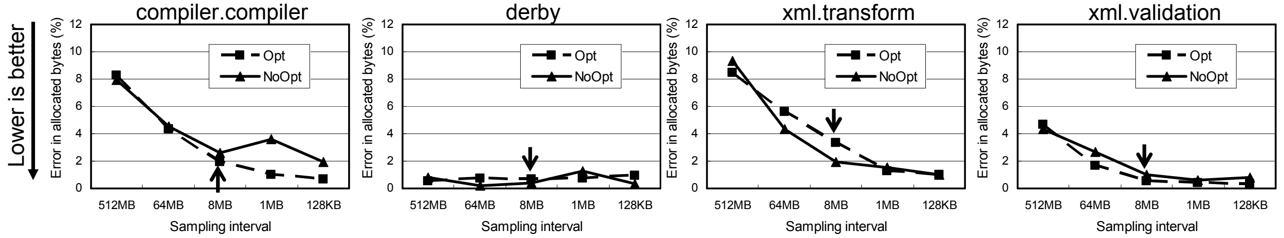

Figure 5. Accuracy: estimation error ratios in the allocated bytes of write-only objects for 4 representative allocation-intensive benchmarks, comparing the 10 combinations of 512-MB/64-MB/8-MB/1-MB/128-KB and Opt/NoOpt (= with/without the overhead reduction techniques in Section 4.2). Black arrows point to 8-MB/Opt, which we chose as the best configuration to balance accuracy with overhead.

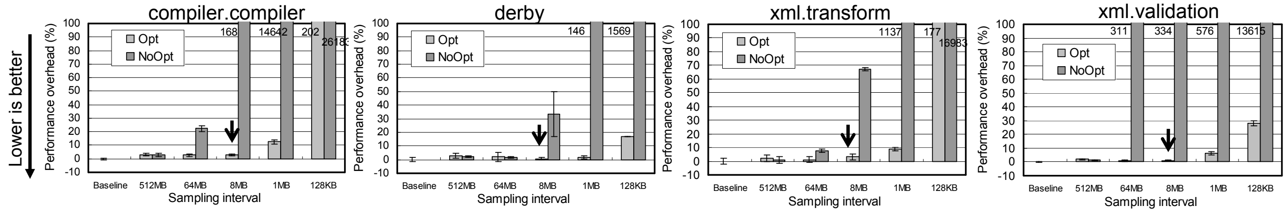

Figure 6. Performance overhead: relative overheads in the 4 benchmarks with 90% confidence intervals, comparing the 10 configurations. Black arrows point to 8-MB/Opt, which was the best configuration to balance accuracy with overhead.

the interval became shorter, the adaptive unbarrierization had larger effect on the performance.

With the 8-MB/Opt configuration, an object access optimization needs to tolerate at most 10% errors in the write-only ratios at each allocation site, as described in Section 5.5. Suppose we use a speculative optimization that compresses the objects allocated at sites whose sampled objects have been 100% write-only. In the xml.transform benchmark, for example, the estimation error ratio of 8-MB/Opt is 3.3% for write-only objects, as shown in Figure 5. Therefore, that speculative optimization should provide a benefit even when 3.3% of the objects allocated at the sites are actually not write-only.

5.7 Comparison with previous techniques

Figure 7 is for the comparison between simulated Bursty Tracing, Instrumented Barrier Profiler (8-MB/Opt), Barrier Profiler (8-MB/NoOpt), and Barrier Profiler (8-MB/Opt) for the estimation error ratios of write-only objects. The geometric means for the profiles of immutable objects and non-accessed bytes are shown on the right side. The upper graph presents the results of the allocation-intensive benchmarks, and the lower graph is the non-allocation intensive benchmarks.

It is difficult to precisely measure the estimation errors of Bursty Tracing without fully implementing it. We took advantage of the mechanism to collect the full traces described in Section 5.5. We still sampled all of the allocated objects, but instead of always profiling all of the accesses to the sampled objects, we collected the profiles only during the instrumented period. Moreover, we only profiled the accesses to the objects sampled during the instrumented period. Instead of counting method entries and loop back-edges as the original Bursty Tracing does, we counted the number of accesses to any of the sampled objects. We assumed a sampling rate of 1/201, which is the least frequent rate tested in [8]. We tried 20000:100, 200000:1000, 2000000:10000, and 20000000:100000 as the ratios between the checking and instrumented period, and found 2000000:10000 to be the most accurate.

The accuracy of Instrumented Barrier Profiler is the same as that of Barrier Profiler if their sampling intervals and Opt/NoOpt choices are the same. To fairly compare their performance overheads, we used 8-MB/Opt for Instrumented Barrier Profiler too.

As shown in Figure 7, Bursty Tracing is not useful as an object access profiler. The large errors were due to profiling only a subset of the accesses to a sampled object. Increasing the instrumented period did not necessarily mean reducing the errors, because it also increased the checking period and failed in sampling at important allocation sites. Barrier Profiler (8-MB/Opt) was close to Barrier Profiler (8-MB/NoOpt). Its estimation error ratios were worse by 0.1%, 0.6%, and 0.3% in write-only objects, immutable objects, and non-accessed bytes, respectively, on the average of the allocation-intensive benchmarks. We believe these levels of errors are acceptable for object access optimizations. In some non-allocation-intensive benchmarks, the error ratios were almost zero, because they allocated few write-only objects.

Figure 8 compares the relative overheads of simulated Bursty Tracing, Instrumented Barrier Profiler (8-MB/Opt), and Barrier Profiler (8-MB/NoOpt and 8-MB/Opt). To estimate the overhead of Bursty Tracing, as explained in Section 5.2, we measured the overhead of the full instrumentation that inserted profiling code at every heap access. The full instrumentation was 6-10 times slower than the baseline. We assumed a sampling rate of 1/201 and the basic checking overhead of 4.9% reported in [1], and calculated the overhead of Bursty Tracing using Equation 4 in [8].

As shown in Figure 8, Barrier Profiler with 8-MB/Opt incurred at most 3.4% runtime overhead in compiler.sunflow among the allocation-intensive benchmarks. Compiler.sunflow caused the largest number of profiled accesses, which resulted in the largest runtime overhead. The average performance overhead of 8-MB/Opt was 1.3% in the allocation-intensive benchmarks, while that of 8-MB/NoOpt was 75.2%. The large performance differences in SPECjvm2008 and some non-allocation-intensive benchmarks were due to decreasing the sampling frequency for large objects. These results show the effectiveness of the overhead reduction techniques.

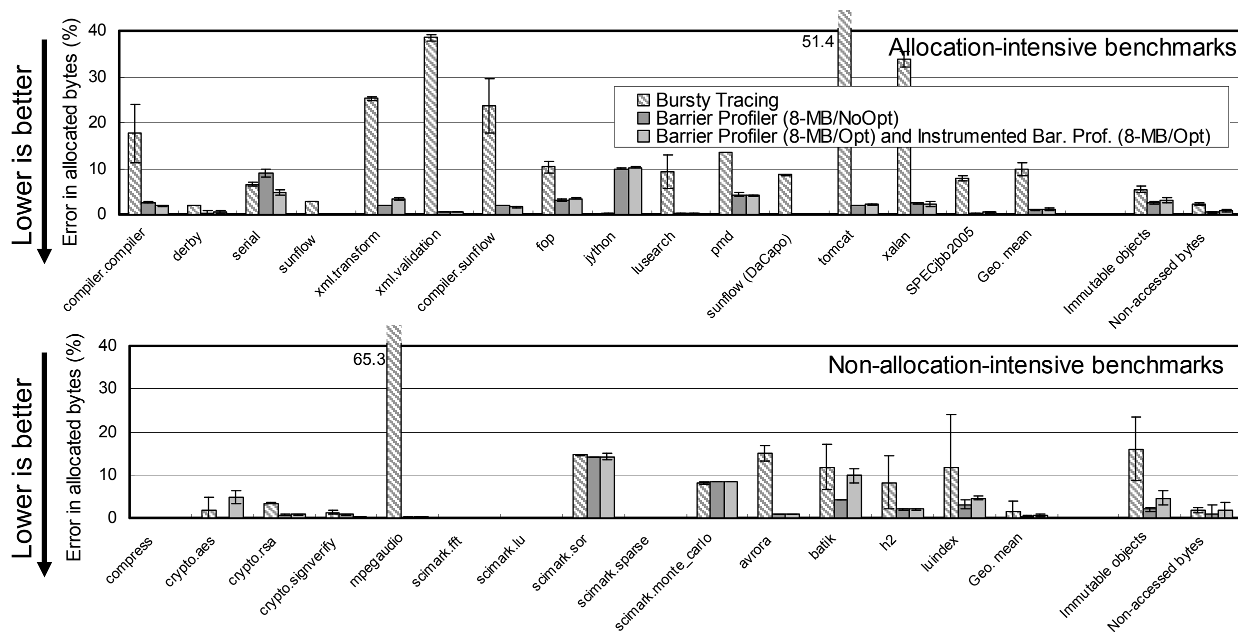

Figure 7. Accuracy: estimation error ratios in the allocated bytes of write-only objects for Bursty Tracing, Instrumented Barrier Profiler (8-MB/Opt), and Barrier Profiler (8-MB/NoOpt and 8-MB/Opt), with 90% confidence intervals. Note that Barrier Profiler (8-MB/Opt) and Instrumented Barrier Profiler (8-MB/Opt) are the same in terms of the accuracy of the profiles.

The overhead of the code instrumentation techniques was larger than that of Barrier Profiler (8-MB/Opt). Bursty Tracing suffered from at most 12.6% and on average 9.2% performance overhead in the allocation-intensive benchmarks, which matches the results in [1]. Even with the Opt configuration, the overhead of Instrumented Barrier Profiler (8-MB/Opt) was 12.4% on average and up to 22.6%. This is 6-9% larger than the instrumentation overhead presented in [3,4], and even larger than that reported in [5]. This is because the instrumentation in [3,4] is inserted only at reference loads. One reason for the difference between [5] and our results may be that the implementation in [5] was done in an embedded JVM, which performs less aggressive JIT optimizations than J9/TR. By removing unnecessary computations, the overhead of the instrumentation is more visible. Another reason appears to be that the instrumentation in [5] was to detect accesses to compressed objects and to decompress them. If one instrumentation dominates the other instrumentation code for the same object in the control flow, then that code can be eliminated. However, for accurate object access profiling, the instrumentation code can only be merged within a basic block.

In the non-allocation-intensive benchmarks, all of the profiling methods suffered from lager runtime overheads than in the allocation-intensive benchmarks because of frequent array accesses. On average, the runtime overhead was 14%, 36%, 666%, and 14% with simulated Bursty Tracing, Instrumented Barrier Profiler (8-MB/Opt), and Barrier Profiler (8-MB/NoOpt and 8-MB/Opt), respectively. Although the 14% overhead is excessive for online profiling, Barrier Profiler is still useful as one of the lowest overhead offline profilers.

In summary, Barrier Profiler achieved sufficiently accurate profiling with low overhead for the allocation-intensive programs, which allows it to be used continuously in the field.

5.8 Results of profiling

So far, we have discussed the accuracy of the profiling, but the profiling results themselves revealed interesting characteristics in the Java benchmarks. First, quite a large number of objects were write-only in some programs. In SPECjbb2005, 99% of the write-only objects were String objects and their associated character arrays. These objects were created for logging, which actually was not enabled during the benchmark run. They are good candidates for removal or compression.

Second, except for xml.validation, more than half of the allocated bytes were immutable objects in the allocation-intensive benchmarks. This matches the report in [26] that many fields in Java object can be considered `final`. Not all of the immutable objects were instances of the classes that are defined as immutable in Java, such as `String`. For 90% of the immutable objects in compiler.compiler and 42% in xml.transform, they were not instances of those known immutable classes. Since the immutable objects cannot be modified, a program should not copy the values of such objects to other objects. Instead, the objects should be shared with a copy-on-write mechanism.

Third, among the allocation-intensive benchmarks more than 20% of the total allocated bytes were never accessed in serial, xml.transform, xml.validation, lusearch, tomcat, and xalan. More than half of the non-accessed bytes were in arrays. Those arrays were buffers allocated with large initial sizes, but only small portions were actually used. Barrier Profiler is useful for tuning the initial sizes to avoid such memory bloat.

6. Online Object Access Optimizations

We demonstrated the usefulness of Barrier Profiler by implementing two new online object access optimizations. One is speculative compression of character arrays, which also uses pointer

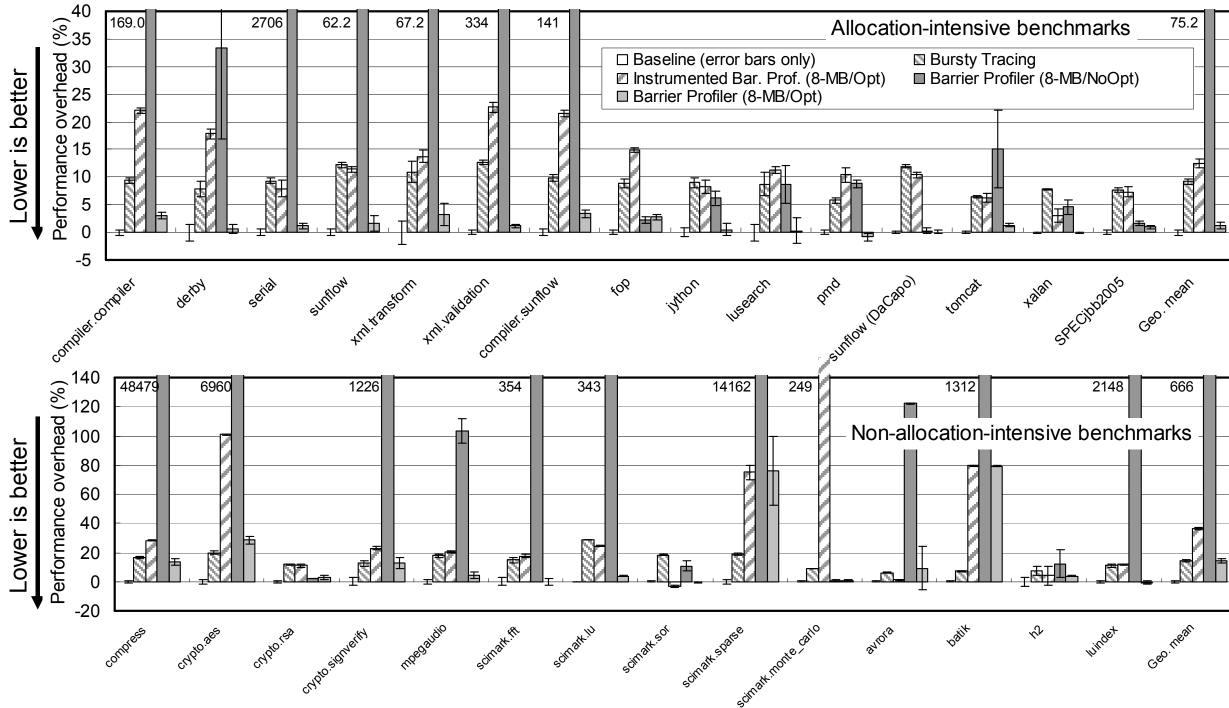

Figure 8. Performance overhead: relative overheads of Bursty Tracing, Instrumented Barrier Profiler(8-MB/Opt), and Barrier Profiler (8-MB/NoOpt and 8-MB/Opt), with 90% confidence intervals. The leftmost error bar of each benchmark is for the baseline.

barrierization as a recovery mechanism, and the other is dynamic adjustment of initial array sizes.

6.1 Speculative online compression of character arrays

As shown in Section 5, many objects were found to be write-only by Barrier Profiler. In SPECjbb2005, character arrays associated with String objects account for most of the write-only objects. These arrays are created by a String constructor `java/lang/String.<init>([C)V`, which allocates a new character array and copies the contents of its parameter array into the new array. Barrier Profiler revealed that there was no write or read to the character arrays outside of the constructor. Therefore, these arrays are good candidates for initialization-time compression.

A 16-bit Unicode character in Java can be compressed by half if it contains an 8-bit ISO-8859 character. A previous compression algorithm [32] that takes advantage of this fact allows direct read and write accesses to a compressed character array by halving the access indices. However, this approach comes at a cost of restricting the format of the compressed character array, so that it can be directly accessed with low overhead. In contrast, since we now expect that the character array will rarely be accessed, the compressed format can be more flexible.

Figure 9 shows the compression algorithm. Figure 9(a) is a simplified version of the original `java/lang/String.<init>([C)V`. We modified the JIT compiler to generate the code in Figure 9(b) for particular allocation sites that always allocate write-only character arrays according to the online feedback from Barrier Profiler. When allocating a character array at Line 1, the JIT code allocates only a half-sized array. A particular bit in the header of the array is set to indicate that it is a compressed character array. The JIT code stores a barrier pointer to the String object (Lines 2 and 3) to protect accesses to the compressed array. It then copies the array contents with compression (Line 4). If the compression fails due to a non-ISO-8859 character, it uses the original code (Lines 5-9). Figure 9(c) illustrates the copying-with-compression algorithm. Each gray box represents a byte, and a pair denotes a Unicode character. Taking advantage of 8-byte load and store instructions, the algorithm handles 8 characters at a time. The first 8 bytes of `data[]` are shifted left by 1 byte, ORed with the second 8 bytes, and stored into `array[]`. At the same time, these two 8 bytes are ORed and masked to check their high order bytes. When an access to the compressed array is detected by a hardware memory exception, the signal handler decompresses the array in the same way as in previous work [5][32]. That is done by allocating another character array of the original size, decompressing the characters into the new array, and storing a forward pointer in the header of the old array.

We implemented the initialization-time compression of character arrays on top of the pointer barrierization framework described in Section 5.1. We observed an 8.6% speed-up in SPECjbb2005. Since the target character arrays are never accessed, we incurred no overhead due to the signal handling. We found that reduced GC time contributed to about 4.5% improvement. The memory allocation rate was reduced by 17%. The other benchmark programs were not improved, but the overhead was not larger than that of Barrier Profiler in Figure 8.

6.2 Dynamic adjustment of initial array sizes

Programmers often over-allocate large arrays as buffers [20,23]. An example is `BufferedReader` in the Java standard library. It allocates an 8-K-element character array in the constructor, but a short input stream will use only the first few dozen elements.

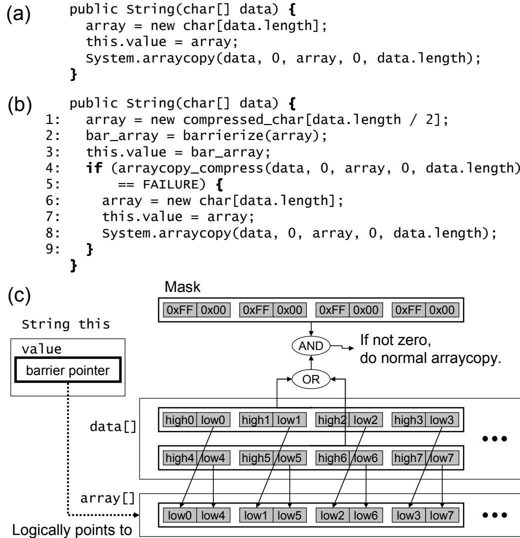

Figure 9. Initialization-time compression of a character array. (a) Original `java/lang/String.<init>([C)V` . (b) Code generated by our JIT compiler. (c) Word-by-word copying with compression.

Another example is `StringBuilder`, which initially allocates a short fixed-length array and extends it as needed. However, even the initial short length can be too long for very small input, and to make matters worse, some programs allocate millions of such wasteful container objects. Those objects are all short-lived, and thus the GC-time truncation of trailing zeros [5] cannot be used. These programming patterns prevail not only in the standard library but also in user-written programs.

Note that it is always safe to specify shorter initial sizes in these programming patterns, though it could be inefficient. Even if the initial sizes are too short, `BufferedReader` just needs to read from its underlying input stream more often, and `StringBuilder` just needs to extend the arrays more times. Although all of the example classes provide programmers with constructors to specify the initial sizes, such interfaces are often not used to simplify the programming. In addition, it is difficult to estimate the best initial sizes statically, because they depend on the input data and the calling contexts of the constructors. To the best of our knowledge, no system has ever efficiently offered dynamic adjustment of the initial array sizes, probably because of a lack of the key technology, a lightweight object access profiler.

We are proposing a new system that combines Barrier Profiler with a new API and a JIT compiler. Rather than using a complex compiler analysis to find the programming patterns in programs, we will provide programmers with a new API to specify the allocation sites of the arrays whose sizes are to be adjusted. The new API will have a set of static methods added to a standard class, in our prototype, `java.lang.System`. Each method receives a default size as a parameter and returns an array of the default size or an adjusted size. For all of the primitive array types such as `byte[]`, `char[]`, …, and `Object[]`, corresponding methods are provided. Figure 10 shows a new method for character arrays, `getCharArrayOfBestSize()`, and its usage in `BufferedReader`. The only change required is to wrap the allocation of the character array with the new API (Line 7). The users of `BufferedReader` do not need to change their code.

```
   public final class System {
1: public static char[] getCharArrayOfBestSize(int defaultSize) {
2:   return new char[defaultSize];
3: }
   ...
   }
   public class BufferedReader {
4: public BufferedReader(Reader in) {
5:   this.in = in;
6:   //this.cb = new char[8192];  // Original implementation
7:   this.cb = System.getCharArrayOfBestSize(8192);
8:   this.length = this.cb.length;
9: }
   ...
   }
```

Figure 10. A new API for the dynamic adjustment of initial array sizes and its usage in `BufferedReader`.

Allocation-intensive benchmarks	Reduction in alloc. rate (%)	Speed-up with 2x the min heap (%)	Speed-up with 1-GB heap (%)
compiler.compiler	-1.2	-5.9	-2.9
derby	5.9	-3.8	-2.0
serial	9.0	-1.8	-1.3
sunflow	-2.1	-3.2	-2.2
xml.transform	-0.3	-6.2	-1.7
xml.validation	-0.7	-3.3	-2.1
compiler.sunflow	-1.8	-7.4	-2.8
fop	-1.2	-3.0	-1.5
jython	5.5	-2.7	-1.4
lusearch	90.6	298.2	36.3
pmd	3.9	-3.4	1.4
sunflow (DaCapo)	-0.8	-3.5	-0.4
tomcat	6.4	-1.0	-0.5
xalan	27.2	17.3	5.9
SPECjbb2005	2.6	-0.3	-0.3
Geo. mean	17.8	7.4	1.3

Table 2. Effectiveness of the dynamic adjustment of initial array sizes. It sped up lusearch and xalan. Overheads in other benchmarks were due to Barrier Profiler.

The default implementation of the new API is just to return an array of the requested size (Lines 1-3), but this implementation is hidden from the programmers and cannot be relied on. Using the non-accessed-byte profile, Barrier Profiler records the offsets of the last accessed bytes of the sampled arrays allocated in the new API. The records are summarized on a per-allocation-context basis. When a context records a sufficient number (10, in our prototype) of samples and the maximum recorded offset is smaller than the default size, the JIT compiler is invoked to inline the allocation context and to embed the maximum offset as a constant in the allocation site, instead of the default size.

We modified `BufferedReader` and `StringBuilder`, and two classes in lusearch and three in xalan to use our new API. All of the modified classes in lusearch and xalan have the same programming pattern as `StringBuilder`. The second column in Table 2 summarizes the reductions in allocation rates for the allocation-intensive benchmarks. The third column shows the performance improvements when running with twice the minimum Java heap that can run each benchmark. The fourth column is for a 1-GB Java heap, which is used by the experiments in Section 5. The maximum improvement was in lusearch: 36.3% with the 1-GB heap, since it allocates many character arrays of 16-K elements, but never uses more than the first 100 elements. Xalan was also improved, especially in the memory-constrained environment. The performance improvement was mainly because of the reduc-

tion in GC frequency. We did not observe performance degradation except for the overhead of Barrier Profiler itself, showing the accuracy of profiling by Barrier Profiler.

7. Conclusion and Future Work

In this paper, we propose a novel low-overhead object access profiler called *Barrier Profiler*, which uses *pointer barrierization* and adaptive overhead reduction techniques. Pointer barrierization converts all of the pointers to certain objects to corresponding *barrier pointers* that point to read-write-protected pages. Unlike previous memory-protection-based profilers, it enables per-object profiling. Barrier Profiler samples objects at allocation sites, performs pointer barrierization, and detects accesses to the sampled objects. It reduces the number of heavy hardware exceptions by using profile-directed adaptive sampling and unbarrierization. Our experimental results showed that in allocation-intensive benchmarks Barrier Profiler provided sufficiently accurate profiles of write-only, immutable, and non-accessed data with 1.3% on average and at most 3.4% runtime overhead. In contrast, code-instrumentation-based approaches suffered from overhead of 9.2% up to 12.6%. Barrier Profiler is the first low-overhead object access profiler that can be run continuously on production systems. Using Barrier Profiler, we implemented two new online optimizations to compress write-only character arrays and to adjust the initial sizes of mostly non-accessed arrays. Both of them are feasible for the first time by using lightweight Barrier Profiler. They resulted in speed-ups of up to 8.6% and 36%, respectively.

In the future, we plan to use Barrier Profiler for detecting last access points and dead stores. We are also interested in investigating the effectiveness of Barrier Profiler on large-scale server-side applications.

Acknowledgments

We would like to thank Peter F. Sweeney and other members in IBM Research – Tokyo and Watson Research Center for helpful discussions. We are also grateful to anonymous reviewers for providing us with helpful comments.

References

[1] Arnold, M. and Ryder, B. G. A framework for reducing the cost of instrumented code. In *PLDI*, pp. 168-179, 2001.

[2] Arnold, M., Vechev, M., and Yahav, E. QVM: An efficient runtime for detecting defects in deployed systems. In *OOPSLA*, pp. 143-162, 2008.

[3] Blackburn, S. M. and Hosking, A. L. Barriers: friend or foe? In *ISMM*, pp. 143-151, 2004.

[4] Bond, M. D. and McKinley, K. S. Leak pruning. In *ASPLOS*, pp. 277-288, 2009.

[5] Chen, G., Kandemir, M., Vijaykrishnan, N., Irwin, M. J., Mathiske, B., and Wolczko, M. Heap compression for memory-constrained Java environments. In *OOPSLA*, pp. 282-301, 2003.

[6] Chilimbi, T. M., Davidson, B., and Larus, J. R. Cache-conscious structure definition. In *PLDI*, pp. 13-24, 1999.

[7] 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.

[8] Hirzel, M. and Chilimbi, T. M. Bursty tracing: a framework for low-overhead temporal profiling. In *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization*, pp. 117-126, 2001.

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

[10] Le, H. Q., Starke, W. J., Fields, J. S., O'Connell, F. P., Nguyen, D. Q., Ronchetti, B. J., Sauer, W. M., Schwarz, E. M., and Vaden, M. T. IBM POWER6 microarchitecture, *IBM Journal of Research and Development*, Vol. 51 (6), pp. 639–662, 2007.

[11] Li, J, Wu, C., and Hsu W. Dynamic register promotion of stack variables. In *CGO*, pp. 21-31, 2011.

[12] Marinov, D. and O'Callahan, R. Object equality profiling. In *OOPSLA*, pp. 313-325, 2003.

[13] Microsoft. Process Address Space. http://technet.microsoft.com/en-us/library/ms189334.aspx .

[14] Mitchell, N. and Sevitsky, G. The causes of bloat, the limits of health. In *OOPSLA*, pp. 245-260, 2007.

[15] Novark, G. Hardening software against memory errors and attacks. Dissertation. University of Massachusetts - Amherst, 2011.

[16] Novark, G., Berger, E. D., and Zorn, B. G. Efficiently and precisely locating memory leaks and bloat. In *PLDI*, pp. 397-407, 2009.

[17] Oracle. HotSpot VM. http://www.oracle.com/technetwork/ java/javase/overview/index.html .

[18] Power.org. http://www.power.org/ .

[19] Rehr, M. and Vinter, B. The user-level remote swap library. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, pp. 164-171, 2010.

[20] Sartor, J. B., Blackburn, S. M., Frampton, D., Hirzel, M., and McKinley, K. S. Z-rays: divide arrays and conquer speed and flexibility. In *PLDI* , pp. 471-482, 2010.

[21] Sartor, J. B., Hirzel, M., and McKinley, K. S. No bit left behind: the limits of heap data compression. In *ISMM*, pp. 111-120, 2008.

[22] Sartor, J. B., Venkiteswaran, S., McKinley, K. S. and Wang, Z. Co-operative caching with Keep-Me and Evict-Me. In *Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures*, pp. 45-57, 2005.

[23] Shacham, O., Vechev, M., and Yahav, E. Chameleon: adaptive selection of collections. In *PLDI*, pp. 408-418, 2009.

[24] Shaham, R., Kolodner, E. K., and Sagiv, M. Heap profiling for space-efficient Java. In *PLDI*, pp. 104-113, 2001.

[25] Skape. Memalyze: Dynamic analysis of memory access behavior in software. *Uninformed Journal*, Vol. 7, http://uninformed.org/?v=7, 2007.

[26] Unkel, C. and Lam, M. S. Automatic inference of stationary fields: a generalization of java's final fields. In *POPL*, pp. 183–195, 2008.

[27] Wilson P. R. Pointer swizzling at page fault time: efficiently supporting huge address spaces on standard hardware. *University of Illinois at Chicago Technical Report* UIC-EECS-90-6, 1990.

[28] Xu, G. and Rountev, A. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, pp. 160-173, 2010.

[29] Xu, G., Arnold, M., Mitchell, N., Rountev, A., and Sevitsky, G. Go with the flow: profiling copies to find runtime bloat. In *PLDI*, pp. 419-430, 2009.

[30] Zhao, Q., Rabbah, R. M., Amarasinghe, S. P., Rudolph, L., and Wong, W. How to do a million watchpoints: efficient debugging using dynamic instrumentation. In *Proceedings of the 17th International Conference on Compiler Construction*, pp. 147-162, 2008.

[31] Zhao, Y., Shi, J., Zheng, K., Wang, H., Lin, H., and Shao, L. Allocation wall: a limiting factor of Java applications on emerging multi-core platforms. In *OOPSLA*, pp. 361-376, 2009.

[32] Zilles, C. Accordion arrays: selective compression of Unicode arrays in Java. In *ISMM*, pp. 55-66, 2007.