Cycler: Improve Tracing Garbage Collection with Real-time Object Reuse

Abstract

Tracing garbage collection (GC) identifies dead objects by scanning the object-graph from the root set references. It is proven to be highly efficient and widely used in commercial managed runtime systems for Java, C# and scripting languages. However, tracing GC cannot identify the dead objects before finishing object-graph traversal. Hence the application cannot reuse the space of the dead objects in real-time when the objects are no longer live. Even there are lots of dead objects, the system has to keep allocating new space for new objects until the object-graph is scanned, which has negative impact on both space efficiency, bandwidth efficiency and cache efficiency. The issue becomes serious with allocation-intensive applications. Some prior work tries to improve tracing GC with compilation analysis to identify the dead objects in compilation time. But they can only work with sub-optimal GC algorithm such as mark-sweep, and/or can only discover a small portion of the dead objects. In this paper we propose an algorithm *Cycler* that can reuse the dead objects in real-time before the object graph scanning. Cycler enhances the existing high-performance tracing GC with reference counting. The novelty of the algorithm is that it does not interfere with the existing optimal tracing GC, so it can effectively improve the overall performance, as long as the runtime overhead of reference counting is well constrained. To control the runtime overhead, Cycler only counts the object references of major types. In this paper, we describe Cycler design and implementation, and also present our evaluations with standard benchmarks.

1. Introduction

Garbage collection (GC) has been widely accepted for automatic memory management to improve the programmers' productivity and software security. Most commercial advanced managed runtime systems adopt tracing GC as their default GC, where the live objects are identified by traversing the object graph from the root references. The unreachable objects are dead and recycled as garbage. Due to its various advantages in design and performance, tracing GC has been investigated thoroughly by the community and evolved into many algorithm variants, such as mark-sweep, semi-space, compacting, etc.

Tracing GC has an obvious drawback. Depending on the object graph traversal for reachability analysis, tracing GC cannot identify the dead objects until finishing the object graph traversal. This means that the space occupied by the dead objects cannot be reused before a collection cycle happens. In a common design, a collection cycle usually happens when the heap space is full or close to be full. At that time, the heap might be mainly occupied by dead objects. SPECJBB2005 benchmark is a good example. We find that, for every running warehouse of SPECJBB2005, only about 2.5M bytes live objects exist at any time point of its steady phase, and the live objects mainly stay in the latest allocated 36M bytes heap space. When new objects are allocated, roughly equal number of old objects become dead at the same time. If a warehouse is given 256M bytes heap, the early allocated 220M bytes are almost all garbage when the heap is full. Up till then, a collection is triggered.

The useless dead object space before a collection cycle has negative impacts. Firstly, the space usage efficiency is low. We observe that, in the submitted SPECJBB2005 scores recent years at SPEC.org website, the heap sizes used are reported bigger and bigger. Enterprises purchase server systems with large RAM size for high throughput, but they probably do not expect the memory be not actively utilized. Secondly, ever-increasing memory footprint before next collection cycle means constant cache misses for object allocations. Consequently and worse, the frequent memory read/write might saturate the bus bandwidth. That seriously limits the applications' scalability with multiple-core systems [1]. Enterprises purchasing multiple-core systems probably do not expect the bus bandwidth bottleneck be caused by the large heap size. Thirdly, if the tracing GC is not concurrent GC, the object graph traversal phase requires the application execution to pause in order to keep the object graph stable. Low memory usage efficiency means frequent application pauses. If the tracing GC is concurrent GC, the runtime overhead of write and/or read barriers degrade the overall system throughput compared to the stop-the-world GC.

It is desirable if the managed runtime system can reuse the dead objects' space in real-time. Then the problems described above can be largely relieved. Real-time object reuse here means two things: First is that the system can identify the objects immediately once they are no longer reachable or needed by the application; second is that the system can reuse the space of the dead objects for new object allocations right after their death. Some prior work tries to improve tracing GC with compilation-time analysis to identify the dead objects [2]. There are two major problems with this approach. Firstly, they can only work with sub-optimal GC algorithm such as mark-sweep, hence the resulted algorithm cannot achieve better performance than current sophisticated tracing GC. Secondly, compilation time analysis cannot discover the dead objects in programs that have complex access patterns.

In this paper, we propose a new scheme for the runtime system to reuse the dead objects in real-time, called *Cycler*. In the scheme, we introduce reference-counting GC as a complement into the sophisticated tracing GC, so that the

tracing GC still works as usual in the garbage collection cycle, while the reference-counting GC works between two collection cycles (though the interval between two cycles can be enlarged due to the object reuse). The novelty of Cycler is that, it does not interfere with the existing advanced tracing GC, so the reference-counting GC only brings benefits as long as its runtime overhead of reference counting is well controlled. The recent work [3] in reference counting GC claims that the runtime overhead can be reduced to be as small as only 2.6%. Cycler tries to further control the runtime overhead by only counting the references of selected object types. Those types are called *major types*. The main contributions of this paper are as follows:

- Investigate the applications' object allocation and collection behavior, and explore the opportunity of object reuse by introducing the concept of major type.
- Design and implement a working algorithm that can best leverage both tracing GC and reference-counting GC, so that the system performance can be improved over the existing sophisticated tracing GC.
- Propose a scheme that can reuse dead objects in real-time between collection cycles without interfering with the existing tracing GC. The object reuse is designed to be only incremental to the tracing GC.
- Evaluate our design with SPECJBB2005, SPECJVM2008, and Dacapo benchmarks. Our data show that the performance of SPECJBB2005 and SPECJVM2008.derby, Dacapo.chart is improved over Apache Harmony default parallel generational tracing GC.

The rest of the paper is organized as the following. Section 2 discusses the related work. Section 3 presents our design of real time object reuse. Section 4 introduces the detailed implementation of the scheme. Section 5 gives our experiment results and analysis. Section 6 concludes the paper and discusses future work.

2. Related Work

There are usually three approaches to automatically judging if the objects are alive or dead: compilation-time analysis, reference counting and reachability analysis (i.e., tracing GC). Compilation-time analysis like liveness analysis or escape analysis [4] tries to identify the objects' live ranges. Reference-counting and tracing analysis are runtime analyses [5]. Reference-counting tries to find the objects that are no longer referenced (i.e., dead) by the application, while tracing analysis tries to find the objects that are reachable (i.e., live) by the application, then the rest are dead. Reference-counting and tracing analysis are naturally complementary. It is interesting to design some scheme that leverages their complementary nature.

Bacon et al [6] points out that tracing GC and reference-counting GC are in fact duals of each other. They show that all high-performance collectors, such as deferred reference counting and generational collection, are hybrids of tracing and reference counting. The work argues that a correct hybridization scheme can be selected based on system performance requirements and the expected properties of the target application. But they do not propose a new hybrid design. We find that tracing GC cannot discover one dead object earlier than another dead object. It can only discover all dead objects at the same time point – when finishing the object graph traversal. On the contrary, reference-counting GC can identify a dead object once its reference count drops to zero. That means tracing GC is very suitable for batch-mode garbage collection, while reference-counting GC is good at real-time garbage collection. They can be hybridized without interfering with each other.

Zhao et al [1] discover the issue of "allocation wall" in Java applications and claim that the issue can be relieved by object reuse. They categorize Java applications as fully scalable, partially scalable, and hardly scalable. They find that the partially scalable applications are also memory-intensive applications, and frequent memory writes due to objects' eviction from the cache are the root cause of low scalability. They manually modify the applications to reuse objects. In this paper, we develop an automatic object reuse mechanism. Our data show that object reuse can not only relieve the issue of allocation wall, but also effectively reduce the batch-mode collection frequency, hence further improve the system performance.

There is previous work [2, 7, 8] employing compilation-time analysis to identify dead objects and trying to work together with the tracing GC. Cherem and Rugina [7] propose an algorithm to identify object variables and object fields that hold unique references, and then instrument the programs with explicit deallocation of individual objects. The algorithm can only work with mark-sweep GC. The work shows performance improvement with a mark-sweep GC. However, mark-sweep GC itself is not widely accepted as the default main algorithm in current advanced runtime systems for its sub-optimal performance compared to copying or moving GCs.

Guyer et al [4] identify dead objects by combining a lightweight pointer analysis with liveness information that detects the time point when short-lived objects die, then insert calls to free function. They conduct experiments by hybridizing their approach with mark-sweep GC and generational GC. They conclude that mark-sweep GC can benefit from the work, but generational GC cannot. They claim that "it is unlikely that any technique can beat the performance of copying generational collection on short-lived objects." In this paper, our work is incremental to existing GC, and can improve the performance of copying generational collection.

Shankar et al [5] propose an enhancement to classic escape analysis with some novel heuristics to decide the scope of inlining and then inline to certain level. Experimental results

show that the approach can identify over four times of non-escape objects as the classic one does, and get an average speedup of 4.8%. The paper doesn't clarify what GC is used in the baseline system. We believe that, since the escape analysis based system allocates objects on the stack, it can work with most algorithms that manage heap objects, including our system.

There is some work recently trying to optimize reference-counting GC. Some approaches are proposed [3, 9] to remove synchronization or to reduce the overhead of synchronization. Some other approaches [3, 10, 11] are proposed to make update operation of reference count less frequent. Harel and Erez [12] argue that the reference-counting overhead can be dramatically decreased and the atomicity requirement can be eliminated. They claim that reference counting becomes a viable option again for modern computing. Joao et al [13] propose an approach to accelerate reference counting with hardware support. Our paper does not mean to contribute further optimization in reference-counting, but to take advantage of it to enable real-time object reuse as an increment to tracing GC. Those optimization techniques can be applied to reduce the overhead of our system.

Blackburn and McKinley [14] explore a hybridization of tracing and reference-counting in a generational GC, with tracing for young objects and reference-counting for mature objects. They can get 2% performance improvement over the baseline GC where mature objects are managed by mark-sweep algorithm. Our approach does not replace the existing collection algorithm with reference-counting, but use it as an incremental enhancement.

In this paper, we count the references of selected objects of certain types, which we call major types. It is an important characteristic of the application behavior. Similar concept is discussed as "prolific types" [15] and a type-based GC is proposed based on it. Yu et al [16] leverage the same property and propose a GC that divides heap space into reuse-space and non-reuse-space, and allocates objects of prolific types into the former where the collection will be triggered more frequently. Experimental results show that the approach performs 2.7%~8.2% on average better than some common tracing GCs except the generational GC. In contrast, our approach actually triggers the collection less frequently, because the object reuse defers the time of heap becoming full; and our work can achieve better performance than the generational GC.

3. Cycler design for real-time object reuse

Since tracing GC is dominant in current commercial managed runtime systems, we expect Cycler be designed with following conditions: 1) the existing advanced tracing GC in the system is not replaced; 2) the existing tracing GC is virtually not impacted; 3) the real-time object reuse is only an incremental enhancement that can also be disabled. These rules guide the Cycler design and its evaluation.

Real-time (also called on-the-fly in some literatures) object reuse means to recycle the object once it is dead. As we have discussed in last section, real-time object reuse surely cannot happen at collection time, which recycles dead objects in batch-mode. Real-time object reuse must happen in application execution time (i.e., mutation time). To enable the mutation time object reuse, we need solve three fundamental problems:

a) identify the objects' death timely;

b) mange the space of the dead objects; and,

c) enable dead objects space reuse in object allocation.

We describe how Cycler solves the problems in following subsections.

3.1 Real-time objects death discovery

To identify the heap objects death in real-time, there are only two ways: compilation-time analysis or reference-counting. As we have discussed, compilation-time analysis has its limitations in working with moving and/or generational GC. In our evaluation with SPECJBB2005, SPECJVM2008 and Dacapo, we find the compilation-time analysis can only discover a small portion of dead objects. For example, the String objects in SPECJBB2005 are intensively allocated and dead very young, but most of them are associated with global data structures and complex call graph, which makes the compilation-time analysis difficult. Cycler chooses reference-counting for real-time death discovery.

Different from classic reference-counting, Cycler only counts the references of selected objects with following benefits:

1) The runtime overhead of reference-counting can be constrained.

2) The existing advanced GC can still collect garbage as usual. It is unaware of the existence of the reference counting.

3) Cycler only identifies the death of the objects that are allocated after a collection and die before the next collection. Cycler never tries to handle the objects that can survive a collection; instead it leaves them to the existing tracing GC. This is a clever design choice, because the tracing GC is good at discovering live objects with reachability analysis, while reference-counting is good at discovering dead objects timely. By counting only the objects that never survive a tracing collection, the two mechanisms work together seamlessly.

4) Cycler only counts the references of one or two types that are prolific. This is not only to control the runtime overhead, but also makes the object reuse easy, because new object can be allocated directly in the dead body of same type object. There is no space fragmentation and it also saves the time installing the type metadata (i.e., vtable pointer).

To select the objects for reference-counting, we propose the concept of *major type*. Major type of an application refers to the class that satisfies following criteria:

i) *Alloc_Percentage*: Of all the objects allocated in the application, the number of the objects of a major type accounts for a significant ratio, e.g. 10% or more;

ii) *Surv_Percentage*: Only a small portion of the objects of a major type can survive a tracing collection, e.g. 10% or less.

Expressed formally, *Alloc_Percentage* of a type is the ratio of the objects allocated for the type to all the allocated objects between two collections (after *last_collect* and before *curr_collect*), as shown below.

Alloc_Percentage(type) = Alloc_Size(type)/ Total_Alloc

Here *Alloc_Size(type)* represents the number (or size) of the allocated objects of the *type* between two collections; *Total_Alloc* represents the number (or size) of all the allocated objects between the two collections.

Surv_Percentage of a type is the ratio of the objects of the type surviving a collection to all the objects of the type before the collection:

Surv_Percentage(type) = Surv_Size(type, curr_collect) / (Alloc_Size(type) + Surv_Size(type, last_collect))

Here *Surv_Size(type, collect)* represents the number (or size) of the live objects of the *type* after collection *collect*;

For Cycler's purpose, *Alloc_Percentage* is the higher the better, and *Surv_Percentage* is the lower the better. Since common applications usually spend most of the execution time in loops, and the types of the allocated objects in those hot loops are limited, they can be the good candidates of major types.

To verify the major type concept, we investigate the object behavior of the String type and char[80] type in SPECJBB2005. As shown in Figure 1, along with the application's execution, the size (and number) of allocated objects increases linearly. At the same time, the size of allocated objects of String type and char[80] type increases linearly as well, taking about two third of all the allocated objects. However, the size of live objects of the two types does not change. It keeps as a constant at only about 2M bytes throughout the execution. That is, the *alloc_percentage* of two types is 67% by size, and the *surv_percentage* of them depends on the heap size. With heap size of 256M bytes, only 1.2% (by size) of them can survive a collection.

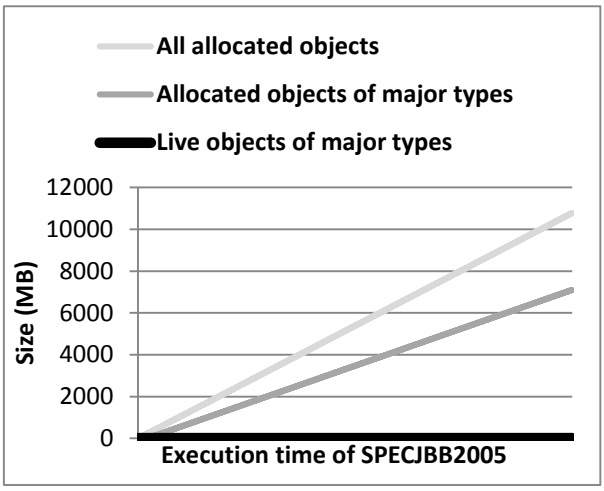

Figure 1: Object behavior of major types in SPECJBB

Major types can be obtained through profiling. With the statistic information on allocated objects before the collections and live objects after the collections, we can get the type distribution of the objects allocated and the survivor ratio of each type. To profile the major types, the application needs to go through several collection cycles to get stable *alloc_percentage* and *surv_percentage*. More details and data are given in following sections.

Once the major types of an application are identified, the runtime system can instrument the program with reference-counting of the major types. Note it is sometimes possible for the programmer to identify the objects' death and manually code to reuse them. However, it is not always feasible in case of the applications that have complex logic. Manual reuse also depends on the programmer to manage the dead objects space at source code level, which is not always possible with high-level languages. Especially when the objects are passed as arguments to library method invocations, we find it is hard to know if the arguments still hold valid live references, and if the live references point to newly allocated objects in the library.

3.2 Dead objects space management

Most commercial tracing GCs use moving collectors. The surviving objects are moved to reduce space fragmentation and improve data locality. The moving collection is essential to support the bump-pointer allocation in thread-local memory block, where the new objects are allocated in a contiguous space by incrementing the allocation pointer linearly.

It is easy to reuse dead objects space in mark-sweep GC because the recycled objects can be put back into the managed free space. But it is not easy to reuse them in a moving GC that has bump-pointer allocation. The difficulty lies in the fact that one cannot simply break the bump-pointer linearity by bumping it back and forth arbitrarily. Cycler solves the problem by not mixing up the dead objects space management with the bump-pointer allocation space. It

manages the dead objects separately with an in-place linked-list. Here "in-place" means every dead object is a node of the list, and the link pointers are stored in the dead object body, which is no longer useful. The object header where the type metadata is stored is untouched. So there is no extra space needed for the reusable space management except the list head pointer, as shown in Figure 2.

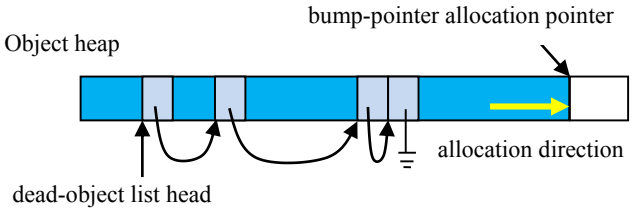

Figure 2: Dead objects space management

In fact, every application thread (i.e., mutator) maintains its own lists of the dead objects, one list for one major type. By operating with the list head, the list node insertion and removal can be as highly efficient as the bump-pointer allocation. When a tracing collection starts, the list head is reset to NULL, no matter whether there are nodes in the list or not. The tracing GC naturally recycles those left nodes. When the collection finishes, the mutator resumes its list maintenance by inserting newly dead objects and removing for new object allocation. We describe the new object allocation in next subsection.

3.3 Dead objects reuse in object allocation

Object allocation is a highly frequent operation in common applications. The allocation sequence is critical to the system performance. Since Cycler is intended to be incremental to existing advanced GC, it tries to keep the object allocation as lightweight as the original bump-pointer allocation.

Intuitively, for each allocation request, Cycler can check if the new object has the same type (or size) with a major type, and then remove a node from the dead-object list for reuse. If the new object is not of a major type or there is no node available in the list, Cycler can proceed with the original bump-pointer allocation. However, this intuitive approach is not efficient because it adds at least one comparison operation in the critical allocation path. Cycler uses JIT (just-in-time compiler) to solve the problem.

During a method compilation, the type information of every allocation is known to the JIT. The JIT can generate appropriate code for the allocation. If the type is a major type, the JIT can transform the allocation to invoke the routine that reuses dead object. Otherwise, it still uses the original bump-pointer allocation routine. In this way, each allocation site in the program only has once compilation-time comparison. There is no more comparison needed at runtime. For the reuse routine, if the dead-object list has no node available at runtime, it falls back to the bump-pointer allocation path. In this way, Cycler hybridizes the allocations of dead objects reuse and original bump-pointer without any penalty.

So far we have described the design idea of Cycler. We expect that the reference-counting effectively identify the dead objects and Cycler effectively reuse the space for new allocations. Then the heap space becomes full more slowly than without Cycler, and the next collection is deferred. In the extreme case, every new allocation request can be satisfied by a dead object reuse. Then the tracing collection will happen very rarely, which virtually results in a reference-counting GC but does not have the drawbacks of a pure reference-counting GC, because it has the flexibility to automatically adapt between a tracing GC and a reference-counting GC. Different GCs run best with different applications' behavior.

4. Detailed implementation of Cycler

To evaluate our design, we implement Cycler in Apache Harmony [17], a product-grade open-source Java Virtual Machine. Harmony is developed with excellent modularity that makes it easy to research new features. The major components relevant to our work are the JIT compiler, garbage collector, and execution engine. Harmony implements many well-tuned GC algorithms, such as partial-forward, semi-space, mark-sweep, move-compact, etc. with generational, parallel and concurrent variants. The default configuration is a parallel generational GC with semi-space for young objects, mark-compact for mature objects and mark-sweep for large objects.

4.1 Cycler infrastructure

Figure 3 illustrates the features that we add into the three Harmony components JIT compiler, execution engine, and garbage collector.

1. At runtime, when a method is first-time invoked, the JIT compiler is triggered to compile it into native code. It takes the profiling results of major types to instrument the compiled code with reference-counting (RC) for selected objects, insert free routine when reference count drops to zero and reuse routine for new allocations of major type objects.
2. When the compiled method is executed at runtime, the reference-counts are updated according to the objects accesses. The free routine is called once a selected object is dead, and the reuse routine is invoked once a selected object is allocated.
3. When the free routine is called, GC inserts the dead object into the reuse list. When the reuse routine is called, GC removes a node from the reuse list and returns it as the newly allocated object.

We give more descriptions in following subsections.

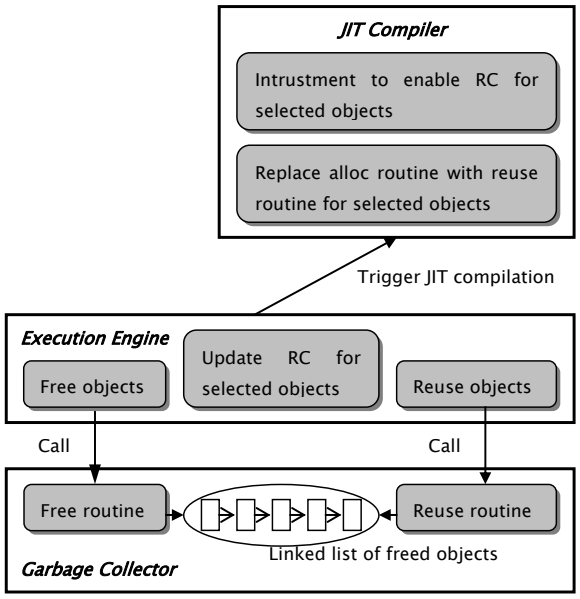

Figure 3: Functionality modification to original Harmony

4.2 Selective reference counting

There are already various approaches proposed by the community on improving the performance of reference counting. For example, Levanoni and Petrank [3] claims that a well-designed reference-counting GC can achieve up to 2.6% negative performance compared with tracing GC. Since Cycler uses selective reference counting, we expect the overhead can be even lower. However, in this paper, we have not implemented those proposed optimizations by the community, since it is not Cycler's purpose to investigate reference-counting optimizations. Cycler implements its plain reference counting with traditional compilation optimizations, so the runtime overhead is expected to be high. On the other hand, if we can achieve good performance in spite of the reference counting overhead, that would demonstrate that the cycler approach is indeed promising.

To support reference counting, the system needs to do following things:

1. Store the reference-counters somewhere to track the objects' references;
2. Add a compilation pass and design the intermediate representation (IR) for reference-counting;
3. Insert operations at appropriate places to update the reference counters;
4. Try to reduce the redundant reference update operations to control the runtime overhead.

We describe them respectively in following text.

4.2.1 Reference counter storage

In Cycler, we choose to store the reference counter in the object itself. An extra word (four bytes) is appended to the original body of selected objects. A reference count may not need a full word, but it is simple and keeps the object aligned. To append an extra word has some implication in the object space management, but it is not an issue in Harmony because the Java object hash-code implementation already appends an extra word when necessary.

Cycler only counts references of thread-local objects. When there is more than one thread accessing same selected object, it is simply discarded, i.e., its RC is not tracked any more. This is to save the cost of atomic operations.

4.2.2 Compilation pass and IR

We add a high-level optimization pass in Harmony JIT called ReferenceCountingPass. It basically inserts operations of calling reference update functions. We extend Harmony high-level IR with the operations in Table 1:

Table 1: RC operations

Opcode	Operands	Semantics
incRC	*tmp1*	Increment RC of the object *tmp1*
decRC	*tmp1*	Decrement RC of the object *tmp1*
updSlot	[*tmp1*], *tmp2*	Decrement RC of the object whose reference is in address *tmp1*, and increment RC of the object *tmp2*

Operation **updSlot** is used when a heap slot (address *tmp1*) is overwritten by a new value (reference *tmp2*).

4.2.3 RC operations instrumentation

The ReferenceCountingPass of Cycler scans a method twice. In the first-pass scanning, it inserts **incRC** for a selected object every time when its reference appears in the stack, and inserts **updSlot** for every time a heap slot is overwritten, except for the method argument and return value. The reference of an argument is hold in the caller's stack frame. The reference of a return value also appears in the callers' stack when current method returns. In the second-pass scanning, ReferenceCountingPass conducts simple liveness analysis for the objects whose RC is incremented by **incRC** or **updSlot**, and then inserts **decRC** for these objects at places right after an object reference is last-time used.

Cycler also needs to update RC for native code. Since JNI specification defines the object manipulation interfaces, we only need instrument following interfaces: SetObjectFieldOffset, SetStaticObjectField, object_clone, and array_copy.

It should be noted that Cycler also instruments the objects whose types are parent types of any major types. The operations will check the real types of the objects at runtime, and only update when they are major types.

4.2.4 RC operations optimization

If without any optimization, the overhead of reference-counting can be unacceptably high. There are two factors in the overhead of reference-counting. One is the number of the operations, and the other is the cost of one operation. Cycler

uses simple optimizations while reducing the overhead dramatically. Here we give two examples.

To reduce the number of the operations, Cycler removes adjacent **incRC** and **decRC** pairs of same object. With this optimization, we can reduce the update operations for the return expressions, and for the assignment expressions to the variables of multi-definition.

To reduce the cost of single operation, Cycler inlines the operations. Harmony has a technique that allows the VM developers to write VM helper routines in Java code, then Harmony JIT can inline the routines at the invocation sites. We write both the reference-count update operations and the free/reuse routines in Java code. They are inlined and optimized by the JIT.

4.3 Reusing the space of dead objects

As discussed previously, Cycler organizes the dead objects into a reuse list. Every time when a new allocation request of the major type comes, the head node of the list is removed and returned. The object header does not need to be refilled for type metadata, but the object body is zeroed.

To support the space reuse, we extend the opcode **decRC** to **decAndtestRC**, so that every time when the RC falls to zero, the object is recycled. In that case, this operation continues the decrement and test operations recursively with the recycled objects' field references, if they have major types.

We also need a dedicated **testRC** opcode, because the redundancy optimizations might remove both **incRC** and **decRC** pairs in a method. But we still need to check the reference-count when it returns from a method call.

To support the object allocation with reuse, we introduce **allocType** for major type object allocations.

Table 2 shows all the finally introduced opcodes by Cycler.

Table 2: All new operations Cycler introduces

Opcode	Operands	Semantics
incRC	*tmp1*	Increment RC of object *tmp1*
decAndtestRC	*tmp1*	Decrement RC of object *tmp1*, test if it falls to 0; If true, recycle it and proceed recursively
testRC	*tmp1*	Test if RC of object *tmp1* falls to 0, and if so, recycle it and proceed recursively
updSlot	[*tmp1*], *tmp2*	Decrement RC of object whose reference is in address *tmp1*, test if its RC falls to 0, and if so, recycle it and proceed recursively; increment RC of object *tmp2*
allocType	*cls1*, *tmp1*	Allocate a new object of class *cls1* by trying reuse first. If no dead object available, falls back to bump-pointer allocation. Let *tmp1* be the returned object.

5. Experimental evaluations and analyses

We evaluate our implementation of Cycler in Apache Harmony. In this section we describe our experimental evaluations and analyses.

5.1 Evaluation setting

We use Harmony default configuration for our evaluation since it is well-tuned [18, 19]. The default GC is a parallel generational GC that partitions the heap into three spaces, NOS (Nurse Object Space), MOS (Major Object Space), and LOS (Large Object Space). NOS is for new object allocation, and LOS is for large object management. MOS is for the surviving objects from NOS. By default, NOS is managed with semi-space algorithm, MOS with mark-compact algorithm, and LOS with mark-sweep algorithm. The sizes of spaces can adaptively adjust according to the application's behavior dynamically. We specify 256M bytes as the default heap size unless otherwise stated.

We use well-known benchmarks SPECJBB2005, SPECJVM2008 and Dacapo (2006 release) for the evaluations. We run the applications with single benchmark thread (e.g., one warehouse with SPECJBB2005) and multiple benchmark threads. Without explicitly stated, the data are with the single thread mode. The computer platform has Intel Core2 Quad CPU with 2.83GHz frequency and 3.23GB RAM.

5.2 Major types of the benchmarks

We collect the major type data for all the applications of the three benchmarks. The data shows that almost all the applications have types that take significant ratio in total allocations. For example, among the 27 applications, 10 have a respective type that takes more than 50% of the total allocation size; another 8 have more than 20% of the total allocation size. In average of all the 27 applications, major types take 47% of the total allocation size.

We also collect how fast the major types become steady in the course of application execution for all applications. The data shows that the pre-steady period takes less than 5% of total execution for one third applications. This means that profiling is feasible to identify the major types. There are a few applications that do not have steady major type from collection to collection. For those applications, full-run profiling might be needed.

Here in this paper, due to length limitation, we only show the data of hree applications as the representatives, i.e., SPECJBB2005, SPECJVM2008.derby, and Dacapo.chart in Table 3 and

Table 4. We choose them because they are most allocation-intensive applications in the benchmark suites, i.e. they allocate most number of major type objects per unit time.

Table 3: Major types of the benchmarks

Benchmark	Major Types	Alloc Perc. (size)	Alloc Perc. (num.)	Surv Perc.
JBB2005	char[80]	54%	19%	~0%
	java.lang.String	13%	35%	10%
derby	java.math.BigDecimal	33%	26%	~0%
	java.math.BigInteger	28%	25%	~0%
chart	java.lang.String	35%	42%	~0%

Table 4: Time needed to get steady major types

Benchmark	Pre-steady collection cycles	Total collection cycles
JBB2005	8	1977
derby	45	704
chart	1	22

5.3 Overhead of reference-counting

We measure the overhead of reference-counting in Cycler. To characterize the reference-counting overhead, we measure the application performance with basic selective RC and optimized selective RC while disabling dead objects recycling and reusing. The results are shown in Table 5.

Table 5: Time overhead of RC

Benchmark	Instrumented types	Basic RC overhead	Optim. RC overhead
JBB2005	char[80]	35.8%	7.3%
	java.lang.String		
derby	java.math.BigDecimal	19.5%	6.5%
chart	java.lang.String	28.9%	13.0%

We can see that the simple optimizations can dramatically reduce the overhead of basic reference counting. On the other hand, the overhead is still too high. We expect to apply the recent work in the community in next step to further reduce the overhead.

It should be noted that we allocate extra four bytes for each instrumented object, so reference counting not only brings time cost, but also space cost. We expect the overhead can be amortized by the space reusing. The estimated space cost is given in Table 6.

Table 6: Space overhead of RC

Benchmark	JBB2005	derby	chart
Overhead	3.4%	3.3%	5.8%

5.4 Benefits of object reuse in single-thread mode

We measure the performance of Cycler by enabling dead objects recycling and reusing. Table 7 shows the results with single-thread mode.

Table 7: Benefits of real-time object reuse

Benchmark	Perf. Improvement	Collection cycles reduction
JBB2005	6.9%	64.2%
Derby	5.2%	31.4%
chart	3.0%	33.3%

Here the performance improvement is computed against the selective RC implementation in last subsection. Collection cycles are the counts of garbage collections. We can see that object reuse does bring benefits, and the collection cycles have been reduced dramatically.

To understand where the benefits come from, we collect data on the object reuse ratios, given in Table 8.

Table 8: Object reuse ratios

Benchmark	Major type	Reuse Ratio (size) of major type objects	Reuse ratio (size) of all objects
JBB2005	char[80]	~100%	54%
	String	71%	9%
derby	BigDecimal	~100%	33%
chart	String	87.5%	31%

"Reuse ratio of major type objects" means that, of all the allocation requests of major type, the ratio of those satisfied by reusing dead objects. The ideal value is close to 100%, i.e. bump-pointer allocation is virtually unused for major type allocations. "Reuse ratio of all objects" is the ratio of reusing in all the object allocation requests. We can see that SPECJBB2005 has more than 60% of object allocations can reuse dead objects. Lots of object reuse effectively defer the triggering of next collection cycle, hence reducing the collection cost.

Besides the benefit in space usage, object reuse also helps to improve the memory performance by reducing cache misses and bus bandwidth consumption. Table 9 shows the results we collect with Intel Vtune [20] performance analyzer.

Table 9: Memory performance of object reuse

Benchmark	Reduction in memory bandwidth utilization	Reduction in L2 cache miss rate
JBB2005	33.3%	25.5%
derby	18.2	21.1%
chart	16.7%	12.8%

Although we get obvious performance gain with object reuse compared to the selected reference-counting implementation. The single-thread absolute performance compared to that of without Cycler is still lower, because the object reuse benefits cannot mask all the RC overhead. There are two reasons: firstly our reference-counting implementation is straightforward without applying the state-of-the-art optimizations; secondly the applications in single-thread

mode are not seriously memory-intensive because the memory system of the platform is designed to sustain multiple-thread computation on multiple cores. Assuming the RC overhead can be controlled to be 2.6%, all the applications can have absolute performance gain even with single-thread mode. Next we investigate the benefits of object reuse with multiple-thread mode.

5.5 Benefits of object reuse in multi-thread mode

We collect the data in Table 10 for SPECJBB2005 running with four warehouses and derby with four benchmark threads. Chart is excluded in the experiment because it is a single-thread application. Memory performance data is also shown in Here the "Absolute performance gain" shows the improvement of Cycler over Harmony default tracing GC. We expect to get more improvement with more sophisticated RC optimizations.

Table 11. We choose four threads because the platform has four cores, thus the data are supposed to be the peak numbers.

Table 10: Benefits with multi-threading applications

Benchmark	Overhead of RC	Benefit of reuse	Absolute perf. gain	Reduced collection cycles
JBB2005	6.1%	9.0%	**2.3%**	19.3%
derby	4.2%	11.7%	**7.0%**	20.9%

Here the "Absolute performance gain" shows the improvement of Cycler over Harmony default tracing GC. We expect to get more improvement with more sophisticated RC optimizations.

Table 11: Memory performance with multi-threading

Benchmark	Reduction in memory bandwidth utilization	Reduction in L2 cache miss rate
JBB2005	7.1%	5.9%
Derby	8.6%	9.5%

As can be seen from the data, compared with single-thread mode, the multi-thread mode gets less reductions in collection cycles, memory bandwidth utilization, and L2 cache miss rate, but Cycler brings much more significant benefits, which indicates that these three factors impact the performance of multi-threading applications more than that of single-threading applications.

The last experiment evaluates whether Cycler can bring benefit with more heap size and more cores. We run SPECJBB2005 and derby on a machine with 4 Intel Dunnington CPUs (24 2.66 GHz cores) and 32GB RAM, with four heap size configurations: 256MB, 512MB, 786MB, and 1024MB. The data are the formal scores reported by the benchmarks.

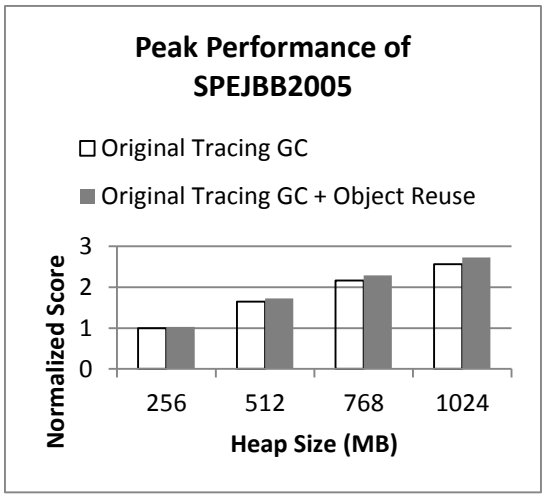

Figure 4 : Scalability under different heap size (JBB2005)

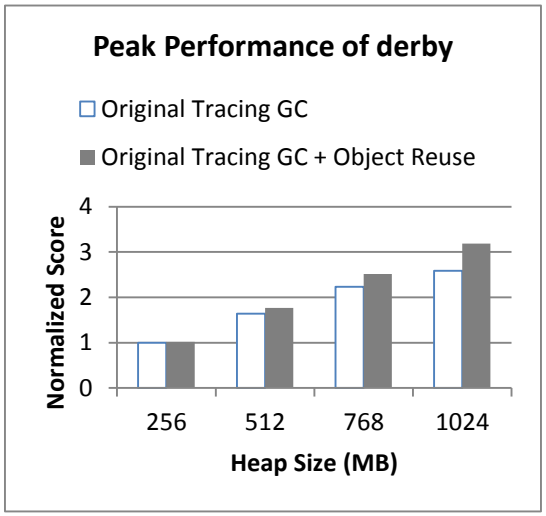

Figure 5: Scalability under different heap size (derby)

As can be seen from the figures, Cycler achieves better scalability under across all the configurations of heap size. In average, Cycler has 3.2% ~ 6.2% better performance for SPECJBB2005 and derby. Besides, we observe that as heap size increases, the advantage of Cycler is more significant. The reason is that for larger heap size, the benchmark performance is more limited by those three factors mentioned previously. Actually Cycler achieves better scalabilities with SPECJBB2005 and derby with more cores than the default GC in our measurement.

6. Summary and future work

In this paper, we investigate the concept of major type, and propose a scheme Cycler to leverage the property by reusing the dead objects of major types in real-time. The novelty of Cycler is that, it does not interfere with the existing advanced tracing GC. We find that tracing GC cannot discover any dead object earlier than another dead object. It can only discover all dead objects at the same time point, i.e., when

finishing the object graph traversal. In contrast to that, reference-counting GC can identify a dead object once its reference count drops to zero. That means tracing GC is very suitable for batch-mode garbage collection, while reference-counting GC is good at real-time garbage collection. Cycler hybridizes these two GCs without interfering with each other. Cycler only counts the references of the major type objects, thus the runtime overhead is smaller than a classic reference-counting implementation. At the same time, the major type objects have low surviving ratio between two collections, which make highly efficient object reuse possible. In our evaluation with SPECJBB2005, SPECJVM2008, and Dacapo, Cycler can get absolute performance improvement with three allocation intensive applications with straightforward RC implementation. We expect that more sophisticated reference-counting implementation can bring more benefit and help to scale the applications with more threads. That is our next step work.

We also need to investigate Cycler with more applications. One major task is to understand if object reuse can benefit less allocation-intensive applications. We also want to combine Cycler with compilation-time analysis to identify the object death, and then Cycler does not need to track the reference counts of all the major type objects. The RC overhead can be further reduced. In this way, a synthesis of compilation-time analysis, reference-counting, and tracing collection can be established that leverage the best results of the memory management community.

References

[1] Y. Zhao, *et al.*, "Allocation wall: a limiting factor of Java applications on emerging multi-core platforms," presented at the Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, Orlando, Florida, USA, 2009.

[2] S. Z. Guyer, *et al.*, "Free-Me: a static analysis for automatic individual object reclamation," *SIGPLAN Not.,* vol. 41, pp. 364-375, 2006.

[3] Y. Levanoni and E. Petrank, "An on-the-fly reference-counting garbage collector for java," *ACM Trans. Program. Lang. Syst.,* vol. 28, pp. 1-69, 2006.

[4] Y. G. Park and B. Goldberg, "Escape analysis on lists," *SIGPLAN Not.,* vol. 27, pp. 116-127, 1992.

[5] P. R. Wilson, "Uniprocessor Garbage Collection Techniques," presented at the Proceedings of the International Workshop on Memory Management, 1992.

[6] D. F. Bacon, *et al.*, "A unified theory of garbage collection," presented at the Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Vancouver, BC, Canada, 2004.

[7] S. Cherem and R. Rugina, "Uniqueness inference for compile-time object deallocation," presented at the Proceedings of the 6th international symposium on Memory management, Montreal, Quebec, Canada, 2007.

[8] A. Shankar, *et al.*, "Jolt: lightweight dynamic analysis and removal of object churn," presented at the Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, Nashville, TN, USA, 2008.

[9] A. Gidenstam, *et al.*, "Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting," *Parallel and Distributed Systems, IEEE Transactions on,* vol. 20, pp. 1173-1187, 2009.

[10] D. F. Bacon, *et al.*, "Java without the coffee breaks: a nonintrusive multiprocessor garbage collector," *SIGPLAN Not.,* vol. 36, pp. 92-103, 2001.

[11] P. G. Joisha, "Compiler optimizations for nondeferred reference: counting garbage collection," presented at the Proceedings of the 5th international symposium on Memory management, Ottawa, Ontario, Canada, 2006.

[12] H. Paz and E. Petrank, "Using prefetching to improve reference-counting garbage collectors," presented at the Proceedings of the 16th international conference on Compiler construction, Braga, Portugal, 2007.

[13] J. A. Joao, *et al.*, "Flexible reference-counting-based hardware acceleration for garbage collection," presented at the Proceedings of the 36th annual international symposium on Computer architecture, Austin, TX, USA, 2009.

[14] S. M. Blackburn and K. S. McKinley, "Ulterior reference counting: fast garbage collection without a long wait," *SIGPLAN Not.,* vol. 38, pp. 344-358, 2003.

[15] Y. Shuf, *et al.*, "Exploiting prolific types for memory management and optimizations," presented at the Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Portland, Oregon, 2002.

[16] Z. C. H. Yu, *et al.*, "Object co-location and memory reuse for Java programs," *ACM Trans. Archit. Code Optim.,* vol. 4, pp. 1-36, 2008.

[17] *Apache Harmony*. Available: http://harmony.apache.org/

[18] X.-F. Li, *et al.*, "A Fully Parallel LISP2 Compactor with Preservation of the Sliding Properties," in *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, ed: Springer-Verlag, 2008, pp. 264-278.

[19] W. Ming and L. Xiao-Feng, "Task-pushing: a Scalable Parallel GC Marking Algorithm without Synchronization Operations," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, pp. 1-10.

[20] *Intel® VTune*. Available: http://software.intel.com/en-us/intel-vtune/