Implementing an On-the-fly Garbage Collector for Java

Tamar Domani Elliot K. Kolodner* Ethan Lewis Eliot E. Salant
Katherine Barabash Itai Lahan Erez Petrank Igor Yanover
Yossi Levanoni

Abstract

Java uses garbage collection (GC) for the automatic reclamation of computer memory no longer required by a running application. GC implementations for Java Virtual Machines (JVM) are typically designed for single processor machines, and do not necessarily perform well for a server program with many threads running on a multiprocessor. We designed and implemented an on-the-fly GC, based on the algorithm of Doligez, Leroy and Gonthier [13, 12] (DLG), for Java in this environment. An *on-the-fly collector*, a collector that does not stop the program threads, allows all processors to be utilized during collection and provides uniform response times. We extended and adapted DLG for Java (e.g., adding support for weak references) and for modern multiprocessors without sequential consistency, and added performance improvements (e.g., to keep track of the objects remaining to be traced). We compared the performance of our implementation with stop-the-world mark-sweep GC. Our measurements show that the performance advantage for our collector increases as the number of threads increase and that it provides uniformly low response times.

Keywords: Java, programming languages, memory management, garbage collection, concurrent garbage collection, on-the-fly garbage collection.

1 Introduction

The concept of automatic reclamation of unused heap storage, or garbage collection, was introduced for LISP in 1960. Since then, garbage collection has been adapted for many other languages, such as Scheme, ML, Prolog, Smalltalk, Eiffel and more recently Java.

The first algorithm for GC was the mark-sweep technique. In this method, the connectivity graph of a program's live objects is determined by tracing from the set of roots for the program, typically the program's stack, registers and static storage. Objects traced are *marked*, and at the end of the trace, any object not marked is unreachable, and can therefore be reclaimed, or collected by the sweep phase. Typically, this is done in a *stop-the-world* manner, where all program threads, or mutators, are halted for the duration of garbage collection. In early LISP systems, the cost of garbage collection was high; some large LISP programs were spending 25 to 40 percent of overtime time executing GC [22], which resulted in slow program execution as well as long pause times.

Java was initially aimed at clients; however, its write-once-run-anywhere promise has led it to being adopted as a programming language for server applications. These applications tend to be highly multithreaded, with tens to even hundreds of threads running simultaneously. Running a stop-the-world garbage collector in this situation would appear inefficient, since during GC only one of the server's processors is utilized, and without good support from the underlying operating system, quiescing, or stopping all threads, can be a time consuming operation. Furthermore, as heaps and their number of live objects become larger, the time an application is stopped for garbage collection increases, and response times become long and uneven.

There are two approaches to deal with keeping all of the processors active:

*IBM Haifa Research Lab. E-mail: kolodner@il.ibm.com.

1. Concurrent collectors: Running the collector concurrently with the mutators. The collector runs in one thread on one processor while the program threads keep running concurrently on the other processors. The program threads may be stopped for a short time to initiate and/or finish the collection.

2. Parallel collectors: Stopping all program threads completely, and then running the collector in parallel in several collector threads. This way, all processors can be utilized by the collector threads.

In this paper we discuss a concurrent collector; in particular, our collector belongs to a subclass of these collectors, which we call *on-the-fly* collectors. (We use Dijkstra's original term *on-the-fly* for such collectors [10].) An on-the-fly collector does not need to stop the program threads simultaneously, not even for the initialization or the completion of the collection cycle. The study of on-the-fly garbage collectors was initiated by Steele and Dijkstra, et al. [30, 9] and continued in a series of papers [10, 18, 5, 6, 23, 24] culminating in the Doligez-Leroy-Gonthier (DLG) collector [13, 12].

The advantage of an on-the-fly collector over a parallel collector and other types of concurrent collectors [3, 16, 28, 7], is that it avoids the operation of stopping all the program threads. Usually, program threads cannot be stopped at any point; thus, there is a non-negligible wait until the last (of many) threads reaches a safe point where it may stop.

On-the-fly collectors also satisfy the requirement for short and even response times. For DLG in particular, the longest time that a thread waits for garbage collection is the time for it to mark the objects directly reachable from its stack (assuming that garbage collection cycles are always triggered soon enough such that the program never runs out of memory).

The drawback of on-the-fly collectors is that they require a write barrier and some handshakes between the collector and mutator threads during the collection. Also, they typically employ fine-grained synchronization, thus, leading to error-prone algorithms and implementations.

1.1 This work

We adapted and extended DLG for use as part of a Java Virtual Machine implementation on a modern multiprocessor without sequential consistency. To support Java we added support for finalization, weak references and intern table collection. To run in our Java Virtual Machine we modified DLG to allow conservative scanning of thread stacks. To allow execution on a multiprocessor architecture where a load from memory is allowed to pass a store from memory [1], we removed from the algorithm its corresponding dependencies on memory access order.

We also added performance improvements: an efficient implementation of the handshake mechanism in the presence of non-Java native code (Java native methods), a color toggle mechanism that simplifies the determination of the color for a newly created object and reduces the amount of work during sweep, and a novel and efficient mechanism to keep track of the objects remaining to be traced.

We designed and implemented a heap manager and associated heap data structures (e.g., color bitmap) that avoid fragmentation and reduce the working set during sweep by allowing objects to be freed without touching them.

We implemented our on-the-fly collector as part of a JDK 1.1.6 prototype on AIX and compared its performance with an implementation of a stop-the-world mark-sweep collector for the same Java Virtual Machine. Our measurements on a multiprocessor show a performance degradation for a single thread. However, for greater than three threads the on-the-fly collector always performs better; the advantage generally increases as the number of threads increase and in some cases is as high as 20%. Our measurements also show short and even response times; when the garbage collector thread is given a priority boost on a multiprocessor (thereby ensuring that it always runs when it has work) the maximum response time for 7 threads or less was 100 ms and the highest measured response time was 540 ms.

A collector similar to the one described in

this paper is part of the production JVM on OS/400 and S/390.

In a related paper[14, 15], we show how to incorporate generations into our on-the-fly collector and compare the performance of the collector with generations to the collector described in this paper.

1.2 Organization

We start with a review of the DLG algorithm in Section 2. We continue with a description of the adaptations for the memory coherency model of the underlying hardware and conservative garbage collection, and the additions to support Java in Section 3. We describe our performance improvements in Section 4 and our heap manager in Section 5. We present pseudo-code for our collector in Section 6. We discuss our performance measurements in Section 7. Finally we conclude and present ideas for future improvements in Section 8.

2 Overview of DLG

The Doligez-Leroy-Gonthier algorithm [13, 12] extends the idea for on-the-fly mark-sweep collection presented by Dijkstra [9] to multiple mutators. In the DLG algorithm, the collector moves through three different phases or statuses (*sync1, sync2, async*) during the GC cycle. The transition from stage to stage can only occur once all the mutators have acknowledged that they are aware of the current collector stage through a handshake mechanism. Mutators require a write barrier for object update and coordination with the collector during object creation. The specific action to be carried out during the update or create is determined by the phase of the collector.

Originally introduced by the Dijkstra [9], a color abstraction is used to indicate the state of an object with respect to collection. Colors are assigned to objects or memory locations as follows:
Blue: Memory which is unallocated.
White: Memory which has not been scanned by the collector during the current GC cycle. Memory left white after the collector has completed tracing is garbage.
Gray: Objects marked by the collector, but whose children may have not been marked.
Black: Objects marked by the collector, whose direct children have also been marked.

Each of the mutator threads and the collector thread have its own status variable. At the start of the GC cycle, the collector sets its status to *sync1*. A mutator acknowledges this change in the collector's status by changing its own status to *sync1*. This is called a *handshake*. A mutator may not handshake while it is updating an object slot or creating an object.

During *sync1*, mutators create objects white. When a slot containing a pointer to an object is updated, the write barrier shades gray both the object referenced by the old value and the object referenced by the new value.

After all mutators have moved into *sync1*, the collector moves to *sync2* and requires the mutators to handshake. The write barrier action and the color chosen for object creation during *sync2* remain the same as for *sync1*; *sync2* acts as a tripwire to guarantee that all memory transactions that started before all mutators transitioned to *sync1* have completed.

Once all threads are in *sync2*, the collector moves to *async*. The trace and sweep occur during *async*. The collector remains in *async* until it begins the next collection cycle.

As each mutator responds to *async*, it marks its roots for the collector to trace. Once a mutator has marked its local roots, it is only required to gray the object referenced by the old value on update. This graying of the old reference by the write barrier continues until the collector completes the trace.

At the beginning of *async* a mutator also begins to color its newly created objects black. This continues until sweep. During sweep the location of the collector's sweep pointer determines the color for newly created objects: an object is created white if it is created in memory which has already been swept, black if not yet swept, and gray if it is created at the point where the collector is sweeping. Starting with the completion of sweep all objects are created white.

3 Required adaptations of DLG

Implementing the Doligez-Leroy-Gonthier algorithm in a JVM required adaptations to support the memory coherency model of the underlying hardware, the Java language, and conservative garbage collection.

3.1 Memory coherency

Explicit synchronization is avoided in several places in the DLG algorithm by assuming that order in which load and store operations appear in the algorithm is the order in which they are actually executed. These assumptions hold for multiprocessors, which support sequential consistency [25, 1]. Many modern multiprocessors, however, use memory consistency models that permit the reordering of memory accesses by the CPU in order to enable hardware performance optimizations such as write buffers. On such machines, the use of explicit system dependent synchronization operations, often called *fences*, is required in order to guarantee the correct order of execution. These are expensive operations, and in many cases, would severely degrade performance.

The platforms on which our on-the-fly collector has been implemented (e.g., OS/400 and S/390) allow load operations to pass store operations. We needed to modify the DLG algorithm to remove these store-load dependencies so as to avoid the need for costly fence operations.

We found two store-load dependencies in the algorithm: in object creation and in keeping track of the objects to be traced. In both cases we found a performance improvement, which replaced the part of the code with the dependency. We describe the first one here and the second one in Section 4.3, where we also describe the multiple mark buffer mechanism that avoids it.

DLG employs an ingenious mechanism for avoiding explicit synchronization between a mutator, when it chooses a color for a newly created object, and the sweep phase of GC. When the mutator creates an object, it colors it black. Then it reads the value of a global sweep pointer. If the address of the object is less than the sweep pointer (sweep always occurs in address order starting at the low addresses), it recolors the object white; if the address is equal, it recolors the object gray. Unfortunately, the correctness of this mechanism depends on the store of the black color preceding the load of the sweep pointer. If the order were to be reversed, the object could stay black and its children could be collected during the next collection cycle. Our color toggle, presented in Section 4.1, also solves the memory coherence problem for object creation.

There are architectures that employ weaker forms of memory coherency, where any pair of memory accesses can be exchanged (assuming no data dependency) [1]. In order to run the on-the-fly collector on these architectures, the load-load, load-store, and store-store dependencies would also have to be removed from the algorithm. This is an item for future work.

3.2 Java language

To support features of the Java language we added support for finalization, weak references and intern table to the on-the-fly collector.

3.2.1 Finalization

In Java, an implementer of a class may define a finalize method. According to the semantics of the language, the JVM must invoke the object's finalize method before reusing the object's memory for another object [17, 27]. The on-the-fly collector can use the same strategy as a stop-the-world collector for dealing with finalization – determine the objects that are ready for finalization during a garbage collection cycle and queue them for later processing by a separate finalizer thread. This requires the addition of a finalize-trace phase after the trace of the live objects has been completed. The interesting point with respect to the on-the-fly collector is that the write barrier does not need to be active during this phase as the mutators will not be able to access the objects being traced.

The head of the list of the objects waiting for finalization is a global root. Updates to the head of the list by the finalizer thread need to

be done with the write barrier just like updates to objects and to any other global root.

3.2.2 Weak references

A weak reference is a reference that does not keep an object from being collected [19]. In Java, a weak reference is encapsulated by a special reference object. In JDK 1.1 there was a single type of reference object, sun.misc.ref [32]. In Java 2, there are three types of reference objects, soft, weak and phantom and associated degrees of reachability [31]. Finalization for Java 2 is also implemented using a special final reference object. These objects can also have associated Reference Queues, where the object is enqueued when the appropriate change to its reachability occurs. We refer the reader to the specification of Reference Objects for more information [31]. We limit our discussion to basic weak references without queues; extensions to support the other reference object types and queues are straight-forward.

A weak reference encapsulates a reference to another object called the *referent*. A weak reference object has two methods of interest: (1) a constructor, which takes an object as a parameter and creates a new weak reference object with the parameter as its referent, and (2) a get method, which returns a normal strong reference to the referent. An object is *strongly reachable* as long as it is reachable through a chain of strong references from a root. An object is *weakly reachable*, when it is no longer strongly reachable, but is reachable through a weak reference. The garbage collector clears a weak reference when it determines that its referent is weakly reachable; if the referent object has a finalize method, it is also becomes eligible for finalization. Accordingly, the clearing of weak references by the collector occurs between the completion of trace and the start of finalize-trace.

Read barrier A mutator using the get method interferes with the trace of the garbage collector. It creates a strong pointer to an object that may be collected otherwise. In particular, if the roots were already marked and the object becomes reachable directly from the roots, then the trace of the collection may not shade it, and the sweep may collect it. Thus, we add a read barrier so that during the trace and until the weak references are processed by the collector, any get of a weak reference object shades the referent. After the collector completes its handling of the reference objects, it is guaranteed that all referents are shaded or the reference objects do not point to them anymore (the pointers were cleared), thus, the read barrier may return without any action.

Our read barrier requires a simple mutual exclusion lock, which we call the *refProcessingLock*, to prevent it from turning a weakly reachable object into a strongly reachable object while the collector processes the weak references. We might enable more concurrency if we were to use a share/exclusive lock instead, e.g., a mutator thread locks it in shared mode when it enters the read barrier (so that many mutator threads can enter the read barrier at once), and the collector locks it exclusive when it processes the weak references. Since the lock is held for a very short time by a mutator thread, we chose to use a simple mutual exclusion lock.

We would like to avoid locking by the read barrier as much as possible so that the read barrier does not reduce the level of concurrency. We observe that a mutator has to shade the referent in the following case: The mutator has marked its roots (i.e., its local collection status is *async*), the collector has not yet completed processing the weak reference objects, and the referent is white.

Accordingly, we avoid locking by the read barrier by checking the color of the referent object, the stage of the collector, and the collection status of the mutator thread. We need to lock only if the referent is white and the collector has started a collection cycle, but has not yet finished processing the weak references. If the referent is already shaded (gray or black), or the collector has completed its processing of weak references (e.g., it could be sweeping or resting between cycles) no locking is required.

Finally, we would like to discuss handshake/cooperate during the execution of the get method. We assume that no handshake/cooperate can occur during the execution by a mutator of a "get" method on a weak reference unless the mutator is waiting

to obtain the reference processing lock. If a handshake/cooperate could occur, the collector could start a new collection cycle (e.g., enter a stage during which graying of the referent is required) after the read barrier check already determined that no graying is necessary. The result could be that the graying of a referent would be missed; thus, leading to a dangling reference.

We show the code for the get() routine in Figure 4.

Processing the weak reference objects During trace the collector links together the weak reference objects that are reachable from the roots, but does not trace through the referents. When the trace is complete, the collector processes the list of weak reference objects as described below.

First, the collector acquires the refProcessingLock in exclusive mode. This will stop mutators from accessing a reference object whose referent is still white. Thus, from this point on in the cycle, the mutators do not change the reachability of an object. They can only touch strongly reachable objects.

Immediately after acquiring the lock, we check that the trace is complete again. This is required because a mutator thread might have just finished a get method, so that the object is colored gray, but not all of its children are shaded.

Next, we make a pass over the list of weak reference objects and clear the ones whose referents are white. After the pass is complete we have the following property: each weak reference object either points to a black object or its pointer is cleared. Since from this point on it is not possible that the get method could access a white object, we can release the weak reference lock and let the mutators use the weak references with no delay.

We show the code for processing weak references in Figure 4.

3.2.3 Intern table collection

The java.lang.String class provides a method for interning of Java strings. If the intern method is invoked on a string, whose value is equal to an already existing string in the pool of interned strings, the string from the pool will be returned. Otherwise, the string will be added to the pool and a reference to it returned.

To accomodate long-running server applications, we would like to allow garbage collection of the pool. This requires coordination between intern and the collector. It is accomplished by using a read barrier similar to the one for weak references, both for the creation of new entries in the intern pool, and for retrieval of existing strings from the pool. The collector removes white strings from the intern table just before sweep.

3.3 Accomodating conservative stack scanning

We had to modify DLG to accomodate conservative stack scanning. This forced us to give up an optimization for the fill operation. We provide details below.

The DLG algorithm differentiates between two type of store operations, the fill operation, which stores a value in a new heap object that is private to a thread, and the update operation, which stores a value in an existing heap object, which may also be referenced by more than one thread. Fill operations do not require use of the write barrier.

The JVM used as a basis for our collector is conservative with respect to the roots, i.e., there is no way to determine if a value in a register or on a stack is a pointer or an integer value which looks like a pointer. This requires that any root value which looks like a pointer must be traced. One side effect of this is that by tracing an arbitrary value on one thread's stack, the collector may get access to an object on another thread's stack while the object is being filled. This could lead to the following scenario:

An object, O, is created white by thread A during *sync2*. Before it can be initialized by the fill operation, thread A is delayed, e.g., by a context switch. Thread B starts scanning its stack as part of its transition to *async*, and encounters an arbitrary value, which it conservatively interprets as a pointer to O. Thread B marks O, sees that it does not have any descendants, and colors it black. Thread A then

resumes, and completes the fill operation by setting a field in O to point to another white object, O', referenced from A's stack. Thread A then pops O' from its stack, leaving the only reference to O' from a black object, O. This will cause O' to be incorrectly collected by the collector.

The fill operation must therefore be done with the same write barrier as for normal object updates. Indentifying opportunities to exploit fill in Java programs is difficult in any case, so this is not a severe drawback.

4 Performance improvements

We added several performance improvements to the algorithm, a color toggle to reduce the cost of sweep and object creation, and multiple mark buffers to keep track of the objects remaining to be traced efficiently. We also chose to implement the handshake mechanism in a way that avoids most of the cost for polling.

4.1 Color toggle

According to the basic mark-sweep algorithm, the color of all black (marked) objects must be reset to white (clear) in preparation for the next GC cycle. We introduce a color toggle similar to previous work [24, 20, 8, 4, 21]. Instead of changing the color for each black object, the color toggle simply changes the interpretation of the colors. There are three main advantages to this modification. First, it reduces the amount of work that the collector must do, since the collector no longer is required to reset black items to white in sweep. Second, it simplifies and speeds up object creation by removing the checks for the location of the sweep pointer. Third, it removes the the store-load dependency in object creation discussed in Section 3.1.

In place of white and black, we employ a *clear color* and a *mark color* respectively. We exchange (toggle) the meaning of the bit patterns used for the clear and mark colors at the start of the collection cycle.

We also introduce an allocation color variable per program thread. A thread assigns its newly created objects its allocation color, regardless of the collector stage or the location of the sweep pointer. A thread's allocation color changes from the clear color to the mark color during its transition to *async*.

Trace colors live objects in the mark color. At entry to sweep, live objects are mark color and garbage objects are clear color. Sweep collects the clear colored objects and recolors them blue. The objects colored mark color automatically become clear color at the start of the next collection cycle when the meanings of clear color and mark color are toggled.

Notice that this transformation preserves the original algorithm. In the orignal algorithm, black objects are set to white by sweep, and then white remains the color for allocations until a thread enters *async* again during the next cycle. Here objects colored mark color are cleared automatically at the start of the next cycle, and a thread starts coloring new objects with the mark color when it enters *async*.

We show the code for the color toggle for the mutator routines (create and cooperate) in Figure 1 and for the collector (clear and sweep) in Figure 2.

4.2 Cooperate mechanism

In order for the collector to progress from one stage to another, all the mutators must handshake with the collector, changing their status to that of the collector. To ensure progress, the mutators must check if their cooperation is required on a regular basis. Typically this is done by polling on method invocation and backward branches [11]. The overhead for polling code may be expensive for short loops and short methods [2].

Also, if a thread is in a blocking system call or executing a native method, it may not cooperate in a timely fashion. One solution is to encapsulate these code sections such that the collector can perform the cooperation for the mutator. DLG refer to this as the mutator delegating to the collector [13, 12]. This is relatively straightforward for *sync1* and *sync2*, just requiring changing the mutator's status. However, *async* is more complicated. In order to carry out the delegated mutator's role for the transition to *async*, the collector has to mark the thread's roots, and guarantee no

new roots are created (and hence potentially skipped over), during the marking. This requires the mutator to save its roots at the start of delegation, and to check that the collector is not executing the handshake on its part before leaving the delegation. This overhead for delegation can also have a serious impact on performance if it is used frequently on short-running sections of code.

An alternative to delegation is to stop, one at a time, each thread that has not cooperated within a certain amount of time. The collector first checks that the stopped thread is not executing object creation, the write barrier or the read barrier for weak references; then it carries out the handshake on behalf of the stopped thread. If the thread is executing in one of these critical sections, the collector restarts the thread and then stops it again, repeating until it can carry out the handshake or the thread has cooperated on its own. Atomicity of these critical sections is ensured by a per thread *cantcooperate* flag, which is set before entering one of these code sections and reset upon exiting. On platforms that do not support efficient thread stopping, signals and a signal handler, which forces the handshake, can be employed instead.

In the actual implementation of our code, we have used a combination of the mechanisms discussed above. Mutator threads poll when they do an expensive allocation, e.g., when an allocation bin needs to be refilled or when allocating a large object (see Section 5). The cost for polling is small compared to the cost of the allocation. We use delegation selectively on functions that are assumed to be long-running (e.g., obtaining a monitor lock). We use signalling as a last resort mechanism.

4.3 Multiple mark buffers

As was shown in Dijkstra's original article, tracing must be carried out until no more gray objects are detected in the heap. This requires repeated scans of the heap, or a data structure that is proportional to the size of the heap, e.g., if a separate color bitmap is used (see Section 5). These repeated scans are expensive. Kung and Song [23] use a double-ended queue to avoid scans, but this solution requires synchronization on inserting objects in the queue when more than one mutator is active.

DLG employs a scan pointer, a dirty flag and a collector mark stack in order to reduce the number of scans [13, 12]. The trace stage of the collector scans the objects in address order to find gray objects, updating the scan pointer as the scan progresses. The collector employs the mark stack to keep track of the children of gray objects that need tracing, emptying the mark stack each time before it continues the scan. The collector sets the dirty flag if its stack overflows. When a mutator executes the write barrier, it also sets the dirty flag if the address of the object it colors gray is less than the scan pointer. Trace repeatedly scans the objects until it completes a scan and the dirty flag is not set. (Notice that the DLG solution requires that the store of the gray color complete before the load of the scan pointer; thus, it suffers from the store-load memory coherence problem mentioned in Section 3.1.)

We have designed and implemented a solution which both eliminates repeated scans of the heap and eliminates, for all practical purposes, the need for synchronization to keep track of the gray objects remaining to be traced. In our solution, each mutator thread is assigned a dedicated mark buffer from a pool of mark buffers. The collector thread has a mark stack.

Each mark buffer has a header with the following fields:

- *lastWrite*: The last position in the buffer which was written by the buffer owner. Only the buffer owner may update it.
- *lastRead*: The last position in the buffer scanned by the collector. Only the collector may update it.

To simplify the presentation we assume here that a single mark buffer is associated with a mutator thread over its lifetime and that it never runs out of space. (In our implementation we keep the mark buffers in a linked list, use an indicator in each buffer to show whether it is in use, and use low cost compare-and-swap synchronization to obtain a new mark buffer for a thread when its previous buffer fills.)

Insertions to a mark buffer are done without synchronization. Upon insertion, the mutator

updates the *lastWrite* field associated with its mark buffer. The collector, upon scanning a mark buffer, will read from the *lastRead* element to the *lastWrite* element, inserting each "popped" element in its mark stack, and updating the *lastRead* field. Tracing is complete when the collector has emptied its mark stack, and made a complete pass through the pool of mark buffers finding that for each buffer *lastRead* is equal to *lastWrite*. The cost of this pass is dependent on the number of threads (mark buffers), rather than the number of objects as in previous solutions.

We show the code for multiple mark buffers in the collector routines of Figure 3.

5 Heap management

Our heap manager uses a scheme similar to that of Boehm et. al. [7]. It differentiates between small, medium and large objects, using a different allocation scheme for each. Allocation of small objects in Java is a frequent operation, and therefore we designed small object allocation for efficiency. Large object allocation is less frequent, and also slower due to required object initialization.

We divide memory into blocks (in our implementation we choose the block size to be the same as the page size). An object is large if its size is greater than half a block. Large object allocations are done in block increments, i.e., large objects are rounded up in size to full block increments.

Small objects are up to 128 bytes and medium objects are between small and large. Small and medium object allocation is done from bins of pre-sized chunks. Each thread has a private vector of small object bins for allocation without synchronization, whereas a global vector of bins of medium objects is shared by all mutators.

Ranges of free memory blocks are held in a binary tree sorted first by size and then by address. When a request for a large object is received, the block manager will select a range of free blocks on a best-fit basis. If there is more than one fit, it selects the one starting at the lowest address. If this range is larger than the requested large object (remembering that large objects are allocated on block boundaries), then it will be split to return an object of the requested size, and a remainder. The remainder will be reinserted into the binary tree. Initially the block tree consists of just a single node of contiguous heap.

We also maintain a color bitmap vector, four bits for every eight bytes of heap memory. Sweep scans this vector without touching the objects. During sweep, if the collector frees an entire block, the block is returned to binary tree, and coalesced with an existing, contiguous range of blocks, if possible. If some but not all objects in the block were freed, then the collector will put the block in a list of partially filled blocks, based on the size of its chunks. Note that queuing partially filled blocks instead of immediately chaining freed chunks into the corresponding allocation bin has two advantages. This solution avoids touching, and potentially incurring a page fault, memory which might never be accessed. In addition for small objects, it saves potential fragmentation by having mutators chain chunks on an as-needed basis, rather than arbitrarily push a chain into a thread's dedicated small object vector.

When a small object bin empties, the collector will attempt to refill the bin by taking a block from the partially filled block bin and linking its available chunks into the allocating bin. If no partially filled block of the correct size exists, the block manager will take a block from the binary tree, chop it into chunks of the corresponding size, and chain the chunks into the allocating bin.

We could readily employ additional techniques for reducing fragmentation and working set size, e.g., by splitting free chunks [26] and by exchanging single free blocks in the middle of the heap for contiguous free blocks at the end of the heap [29].

6 Algorithm details

In this section we provide pseudo-code for our on-the-fly algorithm, including many of the adaptations and performance improvements discussed in earlier sections.

Our presentation of the algorithm is based on that of DLG [13, 12]. One difference with

```
Update(x,i,y):
If (status_m ≠async) then
        MarkGray(heap[x,i])
        MarkGray(y)
    else if (stage = tracing) then
        MarkGray(heap[x,i])
heap[x, i] ← y

Create:
Pick x ∈ free.
color[x] ← allocationColor_m
Return x

Cooperate:
If (status_m ≠ status_c) then
    If (status_m = sync2) then
        For each x ∈ roots:
            MarkGray(x)
        allocationColor_m ← markColor
    status_m ← status_c
```

Figure 1: The mutator routines

```
clear : stage ← clearormark
        exchange values of markColor
                                and clearColor
        weakRefsList.clear()
        Handshake(sync1)
mark: Handshake(sync2)
        allocationColor = markColor
        stage ← tracing
        postHandshake(async)
        mark global roots
        waitHandshake
trace : CollectorTrace()
        stage ← refProcessing
        processRefs()
        stage ← sweeping
sweep : For each object x in the heap:
            if (color(x) = clearColor)
                free ← free ∪ x
                color(x) ← blue
        stage ← resting
```

Figure 2: The collection cycle

DLG is that we separate the handshake into two parts, *postHandshake* and *waitHandshake*, instead of using a second collector thread. Another difference is that we introduce a global *stage* variable to keep track of the collector's progress with respect to the mark-sweep collection cycle. The values for *stage* are from the set *clearormarking*, *tracing*, *refProcessing*, *sweeping*, and *resting*. An ordering is defined on these values such that *clear-or-marking* is the lowest in the ordering and *resting* is the highest.

Figure 1 shows the mutator routines, which are influenced by the collector: the write barrier (update routine), object creation (create routine), and the cooperate routine, which the mutator must call regularly. In the code the notation $heap[x, i]$ denotes slot i of the object at address x.

Figure 2 shows the overall collection cycle. $weakRefsList$ is a list of the weak reference objects encountered during trace. The *clear* operation empties it and the *add* operation inserts an element in it.

Figure 3 shows the routines called by the collector. $markbuffer[m]$ is the mark buffer belonging to mutator thread m; $lastWrite[m]$ and $lastRead[m]$ hold the lastWrite and lastRead values associated with it. *markstack* is the collector's mark stack.

Figure 4 shows the read barrier for weak references and the collector code to process weak references.

7 Experimental results

Our goal is to compare our on-the-fly collector with a stop-the-world mark-sweep collector, to check the response time for on-the-fly collection, and to measure the benefits of our performance improvements. Both collectors were implemented on a JDK 1.1.6 prototype that included a JIT. The stop-the-world collector is the collector that was provided at that time for AIX. The on-the-fly collector employed the improvements and adaptations described in the earlier sections with the exception of the color toggle.

Except where noted the machine used was a 4-CPU 200 MHz PowerPC 640e with 256 MB of main memory running AIX 4.2.1. The measurements reported are based on 10 repeated trials for each data point, averaged. All runs were done on a dedicated machine. Except where noted, all runs were done with the initial heap size set equal to the maximum heap

```
MarkGray(x):
  if (color(x) = clearColor) then
     markbuffer[m][lastwrite[m]] ← x
     lastwrite[m] ← lastwrite[m] + 1

CollectorTrace:
  clean ← FALSE
  while (!(clean))
     clean ← TRUE
     For each m ∈ mutators
        while (lastread[m] < lastwrite[m])
           clean ← FALSE
           lastread[m] ← lastread[m] + 1
           markBlack(markbuffer[m][lastread[m]])
           EmptyCollectorStack()

MarkBlack(x):
If (color(x) ≠ markColor) then
    If (weakReference(x)) then
        weakRefsList.add(x)
        For each pointer i ∈ x except x.referent do:
            CollectorMarkGray(i)
        color(x) ← markColor
else
    For each pointer i ∈ x do:
        CollectorMarkGray(i)
    color(x) ← markColor

CollectorMarkGray(x):
  if (color(x) = clearColor)
     markstack.push(x)

EmptyCollectorStack:
  while (!markstack.empty())
     markBlack(markstack.pop())

Handshake:
postHandshake(s)
waitHandshake

postHandshake(s):
status_c ← s

waitHandshake:
For each m ∈ mutators
    wait for status_m = status_c
```

Figure 3: The collector routines

```
get(r):
x ← r.referent
If (x ≠ null and color(x) = clearColor and
    stage <= refProcessing) then
        Lock( refProcessingLock )
        x ← r.referent
        If (x ≠ null and color(x) = clearColor and
            stage <= refProcessing) then
                MarkGray(x)
        Unlock( refProcessingLock )
else if (x ≠ null) then
        x ← r.referent
return x

processRefs():
Lock( refProcessingLock )
CollectorTrace()
For each weakRef r in weakRefsList do:
    x ← r.referent
    If (x ≠ null and color(x) = clearColor) then
        r.referent ← null
Unlock( refProcessingLock )
```

Figure 4: Weak reference read barrier and processing code

size. There were no effects due to paging.

7.1 The benchmark

We used the Portable Business Object Benchmark (pBOB) for our performance comparisons. pBOB is a kernel of business logic, wrapped in a (self contained) driver that simulates a multithreaded application server environment. pBOB is not a Transaction Processing Council (TM) benchmark, but was inspired by the TPC-C benchmark specification. pBOB is self contained and only depends on the Java Core APIs. pBOB is highly configurable, easy to use, and widely used within IBM for testing and benchmarking JVMs. Due to pBOB's recognised utility, pBOB has been submitted to SPEC for inclusion as a server side benchmark.

tpmBOB, pBOB transactions per minute, is a measure of throughput. Response time is reported by pBOB as the maximum time for a transaction to complete. Thus, it takes into account both the hiccup due to a stop-the-world collector and the slowdown due to an on-the-fly collector. The response times we report is the average of the maximum time reported over 10

runs.

Except where noted otherwise, we used the 1.2 beta version of the benchmark. We typically varied the number of threads from 1 to 20.

7.2 The results

In Figure 5 we compare the tpmBOB throughput for 64 MB and 128MB maximum heap sizes. From Figure 5 we can see that from 3 threads and above the on-the-fly GC improves the tpmBOB from 5% to 10%. The reason for the degradation in tpm with small number of threads is that we pay the cost for on-the-fly collection (the write barrier) without getting its benefit (the stop-the-world collector does little or no collection).

Due to its mechanism governing the triggering of a gc cycle, the on-the-fly collector prototype used for these experiments did not always take full advantage of the memory that it was afforded. The RSS row (resident set size) row provides a performance comparison when both collectors use the same amount of main memory during their runs. We measured the number of main memory pages used by the JVM running the on-the-fly collector for a particular number of threads using a 128 MB maximum heap size. Then we found the maximum heap size that would cause the JVM running the stop-the-world collector to use the same number of main memory pages. Then we measured the throughput for the stop-the-world collector at this heap size. When using the same amount of memory we see that from 3 threads and above the on-the-fly GC improves tpmBOB from 7% to 20%.

In Figure 6 we compare the response time for the GCs. In the figure "o-t-f" stands for "on-the-fly" and "s-t-w" for "stop-the-world". We see that the on-the-fly collector significantly improves the response time except for large numbers of threads. For large numbers of threads the collector thread competes with the mutator threads in order to run on a processor. To overcome this problem we built a version of the collector that increases the priority of the GC thread so that it is greater than the priority of the mutator threads. With this priority boost the collector always gets a processor if it has work to do. With the priority boost the response time of the on-the-fly collector is low and uniform even for large number of threads; it reaches a maximum of 0.54 seconds for 64 MB heaps and 0.34 seconds for 128 MB heaps.

No. threads	4	8	10	20
64m heap	0.7	2.4	1.75	1.05

Figure 8: % TPM improvement for multiple mark buffers

In Figure 7 we compare the heap usage for the two collectors; in particular, the smallest heap in which pBOB completes without running out of memory. The on-the-fly collector requires more memory because it does not move objects to reduce fragmentation. For 20 threads it requires 21% more memory than stop-the-world.

In Figure 8 we compare the on-the-fly collector with and without the multiple mark buffers. Without multiple mark buffers, we used the DLG tracing scheme. The measurements were done on a 4-way 332 MHz PowerPC 604e with 512 MB main memory. The JIT was disabled. The maximum memory size was 64 MB and the initial memory size was 1 MB. The improvement measured was small, but notice that this is an overall application improvement and not just an improvement to GC time.

In Figure 9 we compare the on-the-fly collector with and without the color toggle. The measurements were done on a 4-way S/390 partition running a prototype version of JDK 1.1.6. The pBOB version was 1.0. Again the measured improvement was small; however, it is the overall application improvement.

8 Future directions

Even when using an on-the-fly GC algorithm, for a highly multithreaded program running on a multiprocessor with many processors, the garbage collector thread may be overwhelmed by mutators rapidly creating and discarding objects. Boosting the priority of the collector thread can help, but only up to a certain degree. If the collector is not able to clear garbage at the rate that it is being produced, then eventually the heap will become depleted; the garbage

No. threads	1	2	3	4	5	6	7	8	9	10	15	20
64m heap	-3.1	1.1	4.2	7.0	9.2	9.0	9.0	6.5	9.5	12.9	8.9	
128m heap	-6.8	-0.9	6.6	7.1	6.8	9.2	6.8	6.8	5.5	7.2	9.9	9.8
same RSS	-5.9	-0.06	6.7	11.9	13.6	16.9	15.8	16.4	18.8	20.2	13.6	17.9

Figure 5: % Improvement in TPMs

No. threads	1	2	3	4	5	6	7	8	9	10	15	20
o-t-f 64m heap	0.01	0.02	0.03	0.05	0.08	0.09	0.1	0.11	0.12	0.51	6.59	
s-t-w 64m heap	1.11	1.34	1.54	1.76	1.89	2.07	2.26	2.43	2.7	3.22	5.18	6.59
o-t-f 128m heap	0.01	0.02	0.03	0.04	0.07	0.09	0.09	0.12	0.12	0.13	0.27	1.96
s-t-w 128m heap	1.2	2.18	2.43	2.63	2.83	3.06	3.24	3.44	3.63	3.92	5.73	7.35
o-t-f priorty boost 64m	0.01	0.02	0.03	0.05	0.08	0.09	0.1	0.11	0.13	0.24	0.54	
o-t-f priorty boost 128m	0.01	0.02	0.04	0.05	0.07	0.08	0.1	0.13	0.14	0.18	0.28	0.34

Figure 6: Response time in seconds

collection will be synchronous, so that threads attempting to allocate will have to wait until the collector sweeps memory and begins freeing objects before they will be able to continue. In this case the collector is essentially reduced to working in stop-the-world mode, but with the extra overhead of the write barriers.

A future work direction is to investigate having multiple collector threads. Multiple collection threads will shorten the overall GC cycle. In addition to preventing synchronous GC, this will also reduce the amount of time that the mutators are required to pay the overhead of the write barrier. Thus, it should lead to overall increased throughput. However, running several collector threads in parallel will also diminish the number of processors available to the mutators during collection. This could reduce the amount of application work accomplished during collection and increase the latency perceived by a user for his transactions. Careful tuning will be necessary.

9 Acknowledgments

We thank Alain Azagury, Ray Bryant, John Endicott, Arv Fisher, and Sagi Snir.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consitency models: a tutorial. *Computer*, 29(12):66-76, December, 1996.

[2] Ole Agesen. GC Points in a Threaded Environment. SMLI TR-98-70. Sun Microsystems, Palo Alto, CA, December 1998.

[3] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280-94, 1978.

[4] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66-70, March 1992.

[5] Mordechai Ben-Ari. On-the-fly garbage collection: New algorithms inspired by program proofs. In M. Nielsen and E. M. Schmidt, editors, *Automata, languages and programming*. Ninth colloquium (Aarhus, Denmark), pages 14-22, New York, July 12-16 1982. Springer-Verlag.

[6] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333-344, July 1984.

[7] H. J. Boehm and A. J. Demers and S. Schenker. Mostly parallel garbage collection. *SIGPLAN Notices*, 26(6):157-164, 1991.

[8] J. DeTreville. Experience with Concurrent Garbage Collector for Mudula-2+. Techni-

No. threads	3	4	5	10	15	20
o-t-f	12.00	14.00	17.25	32.75	48.75	64.00
s-t-w	11.25	11.25	17.50	23.50	41.50	52.75

Figure 7: Heap usage in MB

No. threads	1	2	3	4	5	6	7	8	9	10	11	12
	3.09	4.74	6.15	1.12	3.22	3.53	2.87	4.40	1.92	3.92	1.83	1.59

Figure 9: % TPM improvement for the color toggle

cal Report 64, DEC Systems Research Center, Palo Alto, CA, November 1990.

[9] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science*, No. 46. Springer-Verlag, New York, 1976.

[10] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965-975, November 1978.

[11] Amer Diwan and J. Eliot B. Moss and Richard L. Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices. 1992. pages273-282.

[12] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In Conference Record of the *Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, 1994, pages 113-123.

[13] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multithreaded implementation of ML. In Conference Record of the *Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1993.

[14] T. Domani, E. Kolodner, and E. Petrank A Generational On-the-fly Garbage Collector for Java. To appear in the *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Languages Design and Implementation.*

[15] T. Domani, E. Kolodner, and E. Petrank A Generational On-the-fly Garbage Collector for Java. Technical Report 88.385 IBM Haifa Research Lab. Web access: http://www.cs.technion.ac.il/∼erez/gen.ps

[16] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report DEC-SRC-TR-25, DEC Systems Research Center, Palo Alto, CA, February 1988.

[17] James Gosling and Bill Joy and Guy L. Steele, Jr.. The Java Language Specification. Addison-Wesley, 1996.

[18] David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921-930, December 1977.

[19] Barry Hayes. Finalization in the Collector Interface. In *Proceedings of the 1992 International Workshop on Memory Management*, pages 277-298, 1992.

[20] Paul Hudak and Robert M. Keller. "Garbage Collection and Task Deletion in Distributed Systems. In *ACM Symposium on Lisp and Functional Programming*, pp. 168-178, Pittsburgh, PA, August 1982.

[21] L. Huelsbergen and P. Winterbottom. Very Concurrent Mark-&-Sweep Garbage Collection without Fine-Grain Synchronization. In *Proceedings of the 1998 Inter-*

national Symposium on Memory Management, pages 50-54, 1998.

[22] R. E. Jones and R. D. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, July 1996.

[23] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, pages 120-131. IEEE Press, 1977.

[24] L. Lamport. Garbage collection with multiple processes: an exercise in parallelism. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 50-54, 1976.

[25] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. In *IEEE Transactions on Computers*, 28(9):690-691, September, 1979.

[26] Tian F. Lim and Przemyslaw Pardyak and Brian N. Bershad. A memory-efficient real-time non-copying garbage collector. In *Proceedings of the 1998 International Symposium on Memory Management*, pages 118-129, 1998.

[27] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification, Second Edition. Addison-Wesley, 1999.

[28] David A. Moon. Garbage collection in a large LISP system. In Guy L. Steele, editor. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, TX, August 1984, ACM Press, pages 235-245.

[29] Gustavo Rodriguez-Rivera and Michael Spertus and Charles Fiterman. A non-fragmenting nonmoving, garbage collector. In *Proceedings of the 1998 International Symposium on Memory Management*, pages 79-85, 1998.

[30] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495-508, September 1975.

[31] Sun Microsystems. Reference objects. http://java.sun.com/products/jdk/1.2/docs/guide/refo

[32] Sun Microsystems. JDK 1.1.8 Documentation. http://java.sun.com/products/jdk/1.1/docs/index.htm