Thread-Local Heaps for Java

Tamar Domani Gal Goldshtein Elliot K. Kolodner Ethan Lewis
Erez Petrank* Dafna Sheinwald

IBM Haifa Research Laboratory
Mount Carmel
Haifa 31905, ISRAEL

ABSTRACT

We present a memory management scheme for Java based on thread-local heaps. Assuming most objects are created and used by a single thread, it is desirable to free the memory manager from redundant synchronization for thread-local objects. Therefore, in our scheme each thread receives a partition of the heap in which it allocates its objects and in which it does local garbage collection without synchronization with other threads. We dynamically monitor to determine which objects are local and which are global. Furthermore, we suggest using profiling to identify allocation sites that almost exclusively allocate global objects, and allocate objects at these sites directly in a global area.

We have implemented the thread-local heap memory manager and a preliminary mechanism for direct global allocation on an IBM prototype of JDK 1.3.0 for Windows. Our measurements of thread-local heaps with direct global allocation on a 4-way multiprocessor IBM Netfinity server show that the overall garbage collection times have been substantially reduced, and that most long pauses have been eliminated.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms

Languages, Performance, Algorithms

Keywords

Garbage collection, Java, JVM, scalable garbage collection, locality, local garbage collection.

*Computer Science Department, Technion – Israel Institute of Technology, erez@cs.technion.ac.il. Work done in IBM Haifa Research Laboratory.

ISMM'02, June 20-21, 2002, Berlin, Germany.

1. INTRODUCTION

Garbage collectors free the space held by unreachable objects so that this space can be reused for future allocations. In a multithreaded system, several threads may perform program tasks concurrently. If these program threads run independently, allocating and operating their own *thread-local* objects, it is desirable that the garbage collector avoid synchronization between these threads as much as possible. In fact in an ideal scenario where all objects are thread-local, each thread could collect its own garbage without any synchronization at all with the other threads.

In this work, we present a design for thread-local heap management for Java. Our goal is to allow separate memory management work for local objects for each thread, so that collection of local objects is quick and synchronization-free, and to employ full heap collection, which reclaims objects that are shared by the threads, infrequently. A thread-local heap collector should be useful for programs that seldom share objects between the threads.

An important advantage of thread local heaps is scalability. Whereas parallel collection strategies usually don't scale well with the number of processors, thread-local heap collection should scale quite well for applications using mainly thread-local objects.

1.1 Memory manager main ideas

The heap is partitioned so that each thread gets a local area in which it can allocate without synchronization. Using dynamic monitoring via a write barrier, we can determine when an object becomes global, i.e., accessible by more than one thread. When a thread requires space, it may run a local collection and reclaim all its local objects that have become unreachable. Local collections require no cooperation whatsoever with the other program threads. Once in while, a full collection is required. Such a collection stops all threads and collects the whole heap, including global and local objects. A full description of the memory manager appears in Section 3.

1.2 Dynamic vs. static monitoring

One important issue is how to determine if an object is local or global. Several previous papers suggested running escape analysis on the program code to conservatively distinguish between local and global objects at compilation time (e.g. [3, 2, 6, 18, 15]). In this work, we dynamically monitor the locality of objects at runtime. Static analysis has three drawbacks. First, it is conservative, i.e., if there exists any path in the program that causes an object to become global, then the object is treated as global. Second, it only judges objects by their allocation site. If an allocation site produces a million objects and 90% of them are local, the site is identified as global. Third, an object that becomes global long after allocation is considered global as of its creation. Employing dynamic (runtime) analysis, at the expense of a write-barrier, we achieve better monitoring since an object becomes global only when it becomes accessible to more than the creator thread. Our dynamic analysis is also somewhat conservative. We only check that an object becomes accessible to more than one thread and not that it is actually used by multiple threads. Furthermore, if an object becomes global, we never check whether it becomes local again. Although our approach is also cautious, it is far less conservative than static analysis. We believe that more accuracy will require a high execution time overhead and yield only slightly increased accuracy.

1.3 Allocating objects directly in the shared part of the heap

There are some objects which are always global in scope. These objects include threads and classes. Because it is easy to identify these objects at allocation time, a performance improvement is obtained by avoiding the allocation of these objects in the allocating thread's local heap. In the local heap they would negatively impact the efficiency of the thread's local collection. Therefore, these objects are allocated in areas designated for global objects only.

Expanding on this idea yields a further improvement for some applications. In some cases there are certain locations in the application code that allocate exclusively (or nearly exclusively) objects which become global in scope. If these sites can be easily identified at allocation time, then allocating from these sites directly into the global area yields a performance improvement. We denote this operation as direct global allocation. It is possible to carry this analysis further, i. e., to allocate objects directly into the global areas depending on the function call path to the allocation site. We elaborate on this idea and present a preliminary mechanism for it in Section 4.

1.4 Our contribution

Our first contribution is algorithmic. We provide a design for a thread-local memory manager and a way to dynamically monitor local vs. global objects through a write barrier. Next, we suggest the idea of direct global allocation and show how it can be implemented. Finally, we discuss the special considerations for compaction in this scenario.

Our second contribution is implementing the proposed memory management and measuring how it runs for multithreaded Java benchmarks. We have implemented the thread-local heap memory manager and a preliminary mechanism for direct global allocation on IBM's 1.3.0 Java Virtual Machine in interpreter mode and ran measurements on a 4-way multi-processor IBM Netfinity server with 550Mhz Intel Pentium III Xeon and 2GB memory. Note that this version of the JVM is a highly optimized production JVM. However we did not implement the write barrier in the JIT, so we could not measure with the JIT.

Our measurements comparing thread-local heaps with direct global allocation to the base version of the JVM show that the overall garbage collection times have been substantially reduced, and that most pauses have become extremely short. Such benefits usually appear with generational collectors. Here we show that they are also achievable with thread-local heaps. The overall program execution time is basically unchanged. There is a 2-3% gain due to the improvement in collection time and possibly by other factors (such as locality of reference). However, the improvement is offset by the costs of dynamic monitoring (via a write barrier).

1.5 Related work

Several works have investigated detecting local objects via static escape analysis [3, 2, 6, 18, 15]. The goal was identifying, conservatively, at compile time, the objects that are only accessed by their creating thread (and no other thread) and thus can be allocated on the stack of their creating thread. Synchronization can be avoided for these objects.

The only paper that actually suggests a memory manager that uses the static analysis is by Steensgaard [15]. He suggests using the results of a static escape analysis, that identifies sub-heaps rooted at an object (i.e., an object and all of the objects transitively reachable from it) that are only accessed by their creating thread, for memory management. He proposes to allocate a section of the heap for each thread, where the thread allocates its local objects, and to allocate another section of the heap for allocation of shared objects. This memory manager is different from ours. Its local collections are not run independently, but are run only as part of the full collection. In particular, all shared objects are traced and reclaimed by a coordinated full collection of the shared area; after the full collection terminates, each thread continues by tracing its own local heap and reclaiming it. In contrast, our local collectors run independently of other threads, without any cooperation or interruptions to the other threads, and while minimizing the number of full collections. The gain in Steengaard's collector is the additional parallelism of the collection in the final phase. However, the number of full collections is not decreased, and thus, the latency does not benefit over a parallel collector. His results indicate that the separation of objects to local and shared areas is a good direction for further research. Steengaard remarks that his collector could be extended, including extensions similar to the design presented in this paper. However, he does not provide a design of such a collector. Another major difference between these two works is that we use dynamic monitoring of globality/locality rather than static analysis (as discussed in Subsection 1.2 above).

We are not aware of any previous work that reports the effects of thread-local heaps for Java on the throughput and latency of program execution. A thread-local heap memory manager has been used in two commercial JVM's: JOVE [11], and BulletTrain [4]. However, neither of them report on the effect local heaps have on the execution of a program. Doligez, Leroy and Gonthier [9, 8] use thread-local heaps for immutable objects in their implementation of CAML. They present some experimental results for pause times, however, their thread-local mechanism is not appropriate for Java, where almost all objects are mutable.

Our direct global allocation resembles pretenuring for generational garbage collection [5, 10, 1]. There, the memory manager tries to assess which objects have long liftimes, and allocate such objects directly in the old generation. Cheng et. al. [5] use profiling to determine allocation sites that tend to allocate long-lived objects. This profiling is then used to classify allocation sites with which pretenuring is used. Harris [10] uses dynamic sampling to predict the lifetimes of objects. Sampling reduces the cost of obtaining statistics, thus, avoiding the need for a separate profile-gathering phase and allowing pretenuring decisions to be reversed during the run. Finally, Blackburn et. al. [1] improve pretenuring by combining profiling results from multiple applications for common library code with results for objects allocated by the runtime code.

Our direct global allocation uses profiling in a similar manner to classify allocation sites. However, the classification is different. We look for allocation sites that allocate global objects (rather than long-lived objects). Also, we find that the depth of the call chain is critical, whereas pretenuring work has used depth of one in all cases.

1.6 Summary of contents

In section 2 we discuss the mechanism we employed for monitoring locality. In section 3 we discuss memory management issues. This section includes an overall view of the memory manger, discussion of data structures used, the algorithms of the local and global collections, and issues involved in synchronization. In section 4 we discuss the method we used for identifying sites that allocate global objects and also the algorithm we used for direct allocation into shared areas. In section 5 we discuss synchronization between the local and global collections. In section 6 we summarize the JVM compaction algorithm, discuss the issues that thread-local collection introduces and present our solution to these considerations. In section 7 we give a summary and analysis of the results we obtained on two benchmarks. We finish with section 8, which is a conclusion of our work and suggestion for future work.

2. MONITORING LOCALITY OF OBJECTS

In this section we briefly explain how we monitor the state of objects and decide if they are local or global. For each object we keep a flag denoted the *global bit* signifying that it is global. All objects are initially local. However, some of the objects in a Java program are global by nature, in the sense that they can be accessed by any thread in the program. We mark such objects as global before they become accessible to threads other than their creator. A list of these objects include Class objects, Thread and ThreadGroups objects. Note that this list is specific to Java, but the overall idea is not.

When an object cell containing a pointer is modified, a check is made via a write barrier whether a local object becomes a descendant of a global object. If this is the case, then the local descendant becomes accessible by all threads and we should mark it and all its descendants global. This write barrier is executed just **before** the actual update. We use DFS to traverse the descendants of the object. We stress that no synchronization is required for the write barrier. The reason is that before the update is actually performed, the object is local and cannot be accessed by any thread other than its creator. The same holds for all its local descendants. Thus, the DFS is executed locally.

A similar write barrier is executed when updating static variables, creating JNI global references and also interning strings. Objects reachable from these roots are accessible to all threads.

In our implementation, we choose to keep the global bit in a separate bitmap. Each bit in the map represents 8 bytes in the heap. From our partition of the heap to areas (see 3 below) we know that each word in the bitmap can be written by one thread only. Thus, writing the bitmap does not require synchronization either.

We present a brief analysis of the write barrier cost. For each pointer modification we run a light-weight check to determine whether a local object becomes accessible from a global object. If so, a heavier trace marks all the object's children as global. Note that the light check runs much more often then the heavy trace. The check runs during each write barrier execution, whereas the trace runs just once for each global object during its lifetime.

3. THREAD-LOCAL MEMORY MANAGEMENT

In this section we describe memory management with thread-local heaps. We implemented a prototype of our memory management for IBM's 1.3.0 Java Virtual Machine for Windows, which we call the base JVM. The thread-local memory manager adopts the algorithm of the base JVM and adapts it for thread-local allocation. We start with a brief description of memory management in the base JVM.

3.1 Outline of management in the base JVM

The base JVM garbage collection uses a mark-sweep algorithm. Allocation is based on local (and fast) allocation caches [7]. After collection, free chunks of memory in the heap are linked to form a globally accessible free list. Threads allocate local allocation caches from the global free list. The allocation caches are at least 384 bytes. Since the memory in them is contiguous, a thread allocates from its cache by bumping a pointer. When the space in a cache is insufficient for an object allocation, the thread allocates another cache from the free list. Objects larger than 384 bytes are not allocated from local caches. Instead a first fit search is done on the global free list.

3.2 Outline of thread-local management

We divide the heap into large areas of equal size. These areas are on the order of 1 mb in size. The exact size can be set by a command line parameter. Each thread is assigned local areas in which it can allocate without synchronization. As the program executes, a write barrier monitors pointer updates, so that it can keep track of objects that become global. Each thread may run a local collection and reclaim all local objects that have become unreachable. Such a collection requires no cooperation whatsoever with the other program threads. Once in while, a full collection is required. Such a collection collects the whole heap including global and local objects.

More specifically, each thread initially receives an area for the allocation of its objects. When the area is exhausted, the thread may obtain another area from a global pool of areas, or run a local collection (according to the triggering policy). Running a local collection is done independently of any other program thread, and we elaborate on the algorithm in Section 3.5 below. Obtaining an area from the global pool requires thread synchronization. When areas in the global pool are not available, a full collection is initiated. The full collection stops all threads and runs a parallel mark and sweep on the whole heap. We elaborate on the full heap collector in Section 3.6 below.

Threads use local allocation caches as in the base JVM. Instead of there being one global list of free chunks for the entire applicaiton, each area has its own free list of chunks which is used to allocate local allocation caches. When the space in a local cache is exhausted, the thread allocates a new cache from one of its areas. When the thread's areas do not contain enough contiguous free space for a new cache, it executes a local garbage collection. This collection neither traces nor sweeps the global objects. If the space freed in the thread's areas is insufficient to serve the current allocation request, the thread allocates a new area from the global pool of areas. When an area is not available from that pool, a global (also called full) collection is initiated.

During a full collection the global collector stops all threads and uses the original garbage collector of the base JVM. This collector is a mark-sweep collector with a parallel mark phase. One difference between the base JVM (full heap) collection and our global collector is that freed spaces are returned to local free lists rather than to one general free list. Only complete areas that contain no local objects may be returned to the global pool of areas.

On top of this algorithm, we have tested an important improvement in which we allocate objects likely to become global in a designated global area. This yields a noticeable benefit and is described in Section 4 below.

Among other technical difficulties, we note the importance of the coexistence of local and global collections. A global collection cannot start while a local collection is sweeping. Furthermore, a local collection cannot be stopped in the middle for a global collection and then resume at the point it was stopped. A mechanism for the coexistence of local and global collection is described in Section 5. Such a mechanism may be useful in other settings as well. We start with a detailed description of the basic allocator and collector.

3.3 Allocation of objects

Objects are categorized by their sizes in three categories. **Small** objects smaller than the minimum size of a cache; **medium** objects with sizes between the minimum cache and half an area; and **large** objects larger than half an area.

As described in section 3.2 we allocate small objects, which are by far most frequent, by bumping a pointer in the local cache. Medium objects are allocated from the free list by first fit order. Failure to satisfy a medium object request results in a local collection. Afterwards, if the request is still unsatisfied, the thread will request a new area from the global pool. For large objects we allocate one or more (if necessary) consecutive areas from the global pool. The remaining part of the (last) area is inserted into the local free list.

3.4 Data structures

The free list. Each thread has a local free list, organized as a linked list. The list is address-ordered and the allocation strategy is address-ordered first fit. Allocation of caches and medium objects is done from the list. Each entry in the list is large enough to serve as an allocation cache.

The local-free areas. Areas that contain global objects, but no local objects, are kept in an array of linked lists (buckets). They are ordered by the size of the largest contiguous free space in the area. Such areas are obtained for medium-sized objects or cache replacement by best fit strategy when the free list cannot satisfy the request. Access to the buckets is protected by a lock.

The free areas. Areas that are completely free from any objects are kept in a binary search tree. They are sorted by address, and a first fit strategy of allocation is used here as well. When all objects in an area are reclaimed, i.e., it becomes free, then it is inserted into the tree after appropriate coalescing (to satisfy requests for objects bigger than an area). Access to the tree is protected by a lock.

The area table. Finally, we keep a table with an entry for each area containing information about the area (owner thread, largest free chunk in the area, etc.)

3.5 Local collection

We use a mark-sweep collector for local collections. Marking is done only on local objects. Note that it is sufficient to mark from the local roots (e.g., the thread's stack). There can be no pointer from a global object or from global or foreign (e.g., local to other threads) roots to a local objects.

We employ the bitwise sweep strategy used in the base JVM [7], modifying it to take into account global bits. Bitwise sweep takes advantage of the fact that we only need free space large enough to accommodate an allocation cache. The mark bits are kept in a bitmap where each bit represents 8 bytes in the heap. Only the bit corresponding to an object's header is used to indicate the object is marked. During sweep we *or* the mark bits and the global bits. Thus, the local thread may free spaces whose bits in the bitmap are zero. In order to find free space at least the size of a minimum allocation cache size, the bitmap must contain at least two consecutive bytes of zero. Thus, we may go over the bitmap in byte steps looking for a zero byte. When we find one, we move forward in word-sized steps until we encounter one that is not zero. Then we check the exact location where the zeros begin and end, and subtract the size of the last object preceeding the zeros. This method buys us fast reclamation of large free spaces and it ignores small free spaces.

The thread adds the free spaces that it finds to its local free list. This requires no synchronization. In case the sweep discovers an area that is completely free of local objects, or completely free of any object, it puts the area in the respective global data structure of local-free or free areas.

3.6 Global collection

A global collection is initiated when a free area cannot be obtained for an allocation of a medium object, large object, or allocation cache. The global collector collects the full heap, including the local areas. Special care must be taken to handle synchronization between the local collectors and the global collector. We discuss this issue in Section 5 below.

The sweep procedure uses the same ideas as in the local sweep. Namely, it ignores small free spaces and finds larger free spaces quickly.

Freed spaces are put in free lists in the following manner. Areas which are fully freed are inserted into the tree of free areas. Areas containing only global objects are added to the structure of local-free areas. Finally, areas containing local objects are added to the free list of its owning thread. The association of an area with its thread is done through the area table recording for each area the thread owning it.

For the global mark we employ the existing parallel mark-sweep collector from the base JVM. All threads are stopped while the collector works.

4. DIRECT ALLOCATION INTO SHARED AREAS

Usefulness of the thread-local collector relies on the assumption that the great majority of objects remain local through their lifetimes. Applications for which this assumption does not hold may see poor performance as a result of the thread-local collection algorithm.

However, there are ways address this problem so that thread-local collection may work beneficially in such applications. A way which works especially well is the allocation of global objects directly into shared areas which are designated exclusively for global objects.

Direct allocation of objects to global areas requires a priori knowledge that the object will become global. One could use dynamic profiling at runtime to determine the likelihood of the different allocation sites to allocate global objects. Sites with a sufficiently high likelihood to allocate global objects would allocate objects directly into a global area. Implementing low-cost dynamic profiling is complex. Thus, to check the benefits of direct allocation, we built a prototype by profiling during a training run and patching the appropriate *new* bytecodes to do direct global allocation. The technique we used to build the prototype is not suitable for a production JVM.

Direct global allocation reduces the fragmentation with benefits of better utilization of heap space and reduced collection times (especially for sweep). One could think that simpler solutions would work. For example, using a "promotion" procedure during local collection to move all global objects out of the local space. However, in order to move the global objects we would need to stop other threads in order to find and modify all pointers referencing the moved objects. This would require substantial synchronization, eliminating the local nature of local collections. Another solution that might have seemed reasonable is to move global objects out of the local space when they become global (via the write barrier). This solution would not require much synchronization with other threads since all the moved objects are still local. However, there is another problem. Moving objects during the write barrier requires knowledge of all the local references to the moved objects. Finding these references would be an unacceptable cost for the write barrier.

4.1 Implementation

pBOB, one of the benchmarks we tested, has a very high rate of global object creation. It creates global objects at a rate of about 30% of all object creations. We ran pBOB, recording the percentage of global allocations at each allocation site. Sites that had a rate of at least 99% we designated "global sites", that is, sites where we would automatically allocate new objects into a global area.

Having identified the global sites, we needed some way of passing this information to the allocator during runtime. To do this, we stored information in the bytecodes of the methods. We replaced the bytecodes for *new*, *newarray*, *anewarray*, *new_quick* and *anewarray_quick* at global sites with unused bytecodes, which we used to call routines to allocate directly into the global areas.

Bytecode replacement works well when an allocation site is always global. However, there is another category of allocation sites suitable for global allocation. This category is sites that allocate globally conditionally on the sequence of methods calling the allocating method. In these cases a work around can be done, where duplicate methods are created, calls to them are inserted in the first method in the sequence, and the bytecode in the final method in the sequence is replaced. We used this technique in the case of string constructors whose calls to construct arrays of characters were global when the strings themselves were global.

4.2 Global Allocation Algorithm

Using a single global allocation area for all the threads would require synchronization on every global allocation. In order to guard the benefits from the regular allocation algorithm, we created a separate allocation track for global allocations for each thread. The new track is essentially identical to the original allocation path. The only difference is that threads do not receive a full area for global allocations. Instead, in order to optimize heap usage, they receive part of an area. The section of area they receive is reasonably large as to avoid frequent requests for more sections. Other than that, the algorithms for regular and direct allocation to global areas are identical, and even use the same code.

5. SYNCHRONIZING GLOBAL AND LOCAL COLLECTIONS

The coexistence of the global and the local collections requires some synchronization to maintain the integrity of the shared data structures and ensure that heap objects are properly classified as reachable or unreachable. For example, we cannot start a global collection while a local collector is sweeping, because the free list of the local heap is not coherent. On the other hand, we may stop local tracing in the middle to let the full collection start working. However, when the full collection ends the local collector cannot continue from where it stopped without a correction since objects have been reclaimed.

Thus, we need a mechanism that allows the following operations:

1. The global collector is able to check if it is not safe to stop any of the threads for a full collection. If it is not safe to stop a thread for a full collection, we will say that the local thread is executing a *forbidden code segment*.

2. The global collector is able to notify all threads that it wants to start a collection. After getting such a notification, threads will not enter forbidden code-segments.

3. A thread that finishes executing a forbidden code segments is able to tell the global collector it has done so.

4. The collector is able to tell when all threads have finished executing forbidden segments.

We remark that all forbidden code segments appear during allocation or local collection. Code segments where the allocator and collector can simultaneously modify the free or area list are defined forbidden. In terms of efficiency, we set the following guidelines:

1. Wherever possible, we refrain from using locks.
2. Overhead on normal execution of the program threads is minimal.
3. The global collector may be delayed, and pay more in terms of synchronization cost, but the delay is as low as possible.

Note the asymmetric cost preference we make: program threads should not suffer during normal execution since that would have a negative impact on the overall execution time. On the other hand, the global collector is seldom run, and thus, it may pay some synchronization cost without a noticeable change in program execution time. Of course, this cost should be reasonable.

Let us describe the synchronization mechanism. The collector sets a global flag, *gc_pending*, to inform the mutators that it wants to collect. For its part each mutator has a local flag, *gc_ready*, which it cleared upon entering a forbidden code segment. Clearinging the *gc_ready* flag prevents the collector from suspending the mutator. Afterwards the mutator checks the collector's flag. If it is set, the mutator will block on a condition variable which is signaled by the collector at the end of the collection. Threads that are not in a forbidden segment at the start of a global collection are halted by the collector. Special synchronization care must be used to ensure that each thread agrees with the collector on their common state. For example, while the collector discovers that a thread is not executing forbidden code, the collector must be sure that the thread will not start executing forbidden code afterwards.

To describe in more detail the synchronization mechanism we start with the mutators. When a mutator enters a forbidden segment, it starts by clearing the *gc_ready* flag and continues cooperating with the collector by checking if a collection is pending. If it is, the mutator sets the *gc_ready* flag and suspends itself (by waiting on a condition variable). When the collection is complete the collector notifies all waiting threads (by broadcast) and the mutator thread resumes and clears the *gc_ready* flag. Upon leaving the segment the mutator sets the *gc_ready* flag. We have two kinds of forbidden segments, short and long. For a short segment (such as allocation) the mutator cooperates with the collector only at the entrance to the segment. For a long segment (such as a local collection), the mutator cooperates at the beginning and in the middle at some appropriate points. In some cases, such as in the middle of a local collection, the mutator checks whether a full collection has occurred while it was suspended. If this is the case, the mutator leaves the local collection and tries allocating again. Such checks are performed after each cooperation point. The cooperation points are chosen to balance minimizing delaying the global collector against frequent checking of the *gc_pending* flag.

When a global collection is invoked, the global collector sets the *gc_pending* flag. Afterwards it runs a loop until it ascertains that each thread's *gc_ready* flags is set and then it suspends them all. When all threads are suspended, the global collector collects the whole heap. When the global collection is done, the collector resumes the suspended threads, clears the *gc_pending* flag, and signals all the threads that are waiting for completion of the collection by disabling the event.

Note that we have used both a flag, *gc_pending,* and a condition variable since it is much more efficient to check a flag then a condition variable. Recall that one of the main goals is to keep the overhead on the mutators as low as possible during normal execution. Thus, it is desirable to let the mutators check a flag.

Figure 1 presents our synchronization of local and global collections. We would like to point out an interesting race condition that requires the use of a *while* statement (instead of an *if* statement) as the mutator checks the *gc_pending* flag. The race is as follows. Suppose the global collector raises the *gc_pending* flag. A mutator checks the flag, finds it set and suspends itself on a condition variable. Later, the collection ends and the mutator resumes. But before it does anything, it looses control over the CPU and waits. Later, another global collection starts. The *gc_ready* flag is still set because the mutator hasn't had the CPU cycles required to clear it. Now the collector thinks the mutator is safely out of a forbidden code. But the mutator, without checking, may start working on a forbidden segment at the same time that the global collector initiates the collection. This will foil the rest of the execution. Such a scenario is eliminated by the use of the *while* statement. After resuming, the thread will test the *gc_pending* flag again before entering a forbidden segment.

6. COMPACTION

We believe compaction is a worthwhile extension to our algorithm. Compaction affords an opportunity to remove long-lived global objects from the local heaps, which increases the efficiency of local collection. Below we provide a design for incorporation of compaction into the algorithm.

6.1 The basic algorithm

An interesting enhancement to the collector is a compaction mechanism. We use the algorithm currently implemented in the JVM which was presented by Morris [12]. This algorithm requires no extra space, keeps the order of the live objects, and requires only two passes on the heap. It assumes that each object has a pointer to its class, and this pointer is distinguishable from other pointers in the object. In the first pass, the compaction algorithm identifies and modifies all the "forward" pointers in the heap. We call a pointer a *forward* pointer if it points to an address in the heap that is higher than its own address. Other pointers are denoted *backward* pointers. In the second pass we modify all the backward pointers and move the objects.

Both passes proceed from the lowest to the highest address in the heap. During the first pass, we create for each object, A, a linked list of the pointers that reference it. The linked list head replaces A's class pointer which is temporarily stored at the end of the list. As each pointer to A is encountered it is linked at the head of the list. Note that this linked list requires no extra space. During the first pass when we get to an object, we have a list of all the forward pointers to it. We also can calculate where this object will reside after compaction, because we have seen all the live objects before it. Thus, we modify all the pointers in the current linked list and "clear" it by restoring the object's class pointer. As we go on with the pass we create a linked list of all the backward pointers to the object in the same manner. When we are done with the first pass, all forward pointers have been properly modified and each object is associated with a list of all backward pointers that point to it. In the second pass we move each object to its new loca-

```
Global Collector:
        mutex_lock
        set gc_pending
        mutex_unlock
        wait while there are threads whose gc_ready flag is cleared
        suspend all threads
        collect the whole heap
        resume all suspended threads.
        clear gc_pending
        mutex_lock
        signal to all waiting threads (broadcast) /* allowing waiting threads to resume */
        mutex_unlock


Any Other Thread:
        /* Its gc_ready flag is normally set. */
        /* It is cleared upon entrance to no-global-GC segment, and set again upon exit */
        /* When entering a forbidden segment or during checks inside a long forbidden segment do:*/
        while (gc_pending is raised)
                set gc_ready flag
                Wait on condition variable
                Raise a flag to inform the allocator that global gc happened
                mutex_lock
                clear gc_ready flag
                mutex_unlock
```

Figure 1: Global and Local Collection Synchronization Scheme

tion, modify all the backward pointers to reference the new location, and restore the class pointer to the object. By the end of the second pass all objects have been moved and all pointers have been updated.

Remark. Note that if we were using extra space for each object, we could finish all the work in one pass. We could keep for each object a translation between the old and new locations, and move the object to the new location. Afterwards we could update backward pointers using the translation table.

6.2 Modification for our collector

In our setting, we would like to compact local objects in the local areas of the thread. We start with presenting local compaction and then continue with global (full heap) compaction.

Local compaction. Local compaction cannot move global objects or objects which are pinned or dosed (*Pinned* objects are those that cannot move because the JNI has given native code direct access to the contents of the object, e.g., an array. *Dosed* objects are objects that may not move due to the conservative scan of the thread stacks.). Thus, each object is moved to the lowest address possible without moving these objects. Also, we insist that objects do not spill from one area to another (unless they are larger than the size of one area). Thus, an object that moves must fit completely into an area. Recall that global objects do not contain pointers to local objects. Thus, they need not be modified and we disregard them through the passes.

The local compactor moves through the local areas by traversing a linked list of areas that belong to the local thread. Only local objects are checked for forward and backward pointers. We keep a read pointer that goes over the live objects in the heap. As the read pointer passes over the heap we organize free chunks it discovers by putting them in an address ordered linked list. Each object encountered is assigned a new address by employing an address-ordered first fit strategy using the linked list. Note that this tends to keep the order of objects in the heap, but does not guarantee it. However, all the other properties of the algorithm hold, most importantly the end result: the pointers are all correctly updated when compaction is finished.

Global compaction. When we do a global collection, we may move all objects that are not pinned or dosed. However, we have the requirement of keeping local objects in their local areas. Thus, we keep a free list as before for each of the local threads as well as one for global objects. Global objects are moved from local heaps to global areas. The traversal of the heap is done area by area, as the read pointer goes over the heap. Each object encountered may be either global or local. A free space for it is allocated using the free list of the thread that owns the current area or using the global free pointer.

When a new global area is required, an area is taken from the list of free areas of the collector. It is preferable to use areas that are local-free rather than areas that are completely free. In the case a new global area is required but cannot be allocated, we keep the global objects in the current local area of the thread. After compaction is done, freed areas are inserted to the local-free or free areas data structures.

7. RESULTS

7.1 Implementation

We implemented the algorithm on IBM's Java Virtual Machine version 1.3.0 and ran measurements on an IBM Netfinity server with 550Mhz Intel Pentium III Xeon and 2GB memory. We modified the interpreter and ran the original and the modified JVM in an interpreted mode without the JIT. This JVM is highly optimized production JVM.

We ran measurements on two multithreaded benchmarks, pBOB2.0a and Trade 2 on WebSphere under AKStress. The first application is the kernel application of the SPEC2000jbb benchmark, which is described in detail in SPEC web site [14]. We chose to use pBOB over SPECjbb because of the greater flexibility it offers in setting parameters, particularly the number of threads. The second application is the WebSphere [17] webserver application driven by AKStress utilizing Trade2 [16]. We used version 4.0 beta of WebSphere and DB2 version 7.2 as the database. AKStress spawns a configured number of threads to make repeated requests to the WebSphere server for web page retrieval. The Trade 2 benchmark is an IBM internal benchmark that measures the performance of servers running the IBM WebSphere Application Server software. The Trade 2 application was built to emulate an online brokerage firm. It is implemented using a collection of Java classes, Java Servlets, Java Server Pages and Enterprise Java Beans (EJBs).

We implemented a preliminary version of direct global allocation for the pBOB benchmark only. The reason we did not prototype it for Trade2 on WebSphere as well is that our manual modification of the relevant allocation sites is not feasible for the abundant number of relevant sites in WebSphere (when running Trade2). We start by presenting our results running the implementation with and without direct global allocation for pBOB. We then provide the measurements for Trade2 on WebSphere without direct global allocation and infer from the improvement seen on pBOB how direct global allocation might improve performance for Trade2 on WebSphere.

7.2 Measurements

7.2.1 *Full measurements for pBOB*

In this subsection, we report our findings for pBOB. We start with statistics comparing global and local object populations. In Table 1, we report the percentage of allocated **local** objects surviving a global collection (on average), the percentage of allocated **global** objects surviving a collection (on average), and the percentage of allocated objects which are global before and after a collection. All measurements are reported both in terms of the number of objects and the accumulated space they require. We see in Table 1 that during a typical collection more than 99% of the local objects die. A much lower death rate (63-80%) is measured for the global objects. The difference in the death rate explains why the fraction of global objects rises from less than half before a collection to almost 99% after the collection.

Next, we report collection time statistics. In all tables we abbreviate thread-local, TL, and direct global, DG. Note that we must compensate since local collections are run on one processor (out of the four in our machine) and the global collections are run on all the four processors. In order to compensate we can either divide the local collections by 4 (meaning that they only take 1/4 of the CPU overall strength) or multiply the full collection times by 4 (to signify that they run on 4 processors). We chose to divide the local collection times by 4 and this is how the measurements are reported. The collection times so derived can be fairly compared with the times from a run of the garbage collector in the original JVM in which the four CPU's are used by the collector.

heap/	calculated gc time			improvement
threads	base	TL	DG	DG vs Base
128/4	597	1589	582	2.4%
128/12	1352	2536	891	34.1%
128/20	2955	4688	1226	58.5%
256/4	421	1597	366	13.1%
256/12	663	2206	381	42.5%
256/20	947	2252	433	54.3%
256/40	2559	3681	1051	58.9%

Table 2: Garbage collections times (in ms): a summary (pBOB).

The summary of the measurements of collection times is given in Table 2 and the detailed information appears in Table 3. Thread-local heaps with direct allocation of global objects yields a reduction of 2-60% in the overall collection times. Note that direct global allocation is crucial for program sites that mostly allocate global objects. Relying on the thread-local heap scheme alone actually yields worse collection times. Looking into the more detailed measurements in Table 3, we see that the main cost of collection is due to the sweep. In our version of the JVM the cost of sweep is proportional to the number of discontiguous free spaces in the heap. Global objects in the local heaps partition free spaces, decreasing the total number of available bytes while greatly increasing the number of free spaces. Allocating objects directly in a global heap substantially ameliorates the sweep efficiency and provides an improvement in overall collection time.

We now turn to reporting pause times. Since this is a stop-the-world collector, the pause times are the lengths of the collections and the number of pauses is the number of collections. Looking at Table 4 we see pauses for full collections (and original collections) on the order of 100ms. Local collections using thread-local heaps are one magnitude smaller (around 10ms) and direct global allocation reduces the local collection times to approximately 1ms. The reduction of two orders of magnitude in the pauses makes pauses for the local collections unnoticeable for the user. To see the reduction of the number of long pauses (due to full collections) look again at Table 4. We see that when using direct global allocation, the number of long pauses is cut by 38-76% depending on the heap size and number of threads running.

Let us now check the cost of the running the program with dynamic monitoring. Recall that the write barrier that marks objects global just before they become global during program execution. It turns out that the cost of executing the write barrier as a percentage of the overall execution time is between 2.1-2.8% (see Table 5). Running with the JIT this figure most likely will rise sgnificantly. This cost explains why we do not see improvements in overall execution times. Although the collection times have been reduced significantly (sometimes by more than a half), the cost of the

heap/ threads	% surviving locals		% surviving globals		% global before gc		% global after gc	
	obj	byte	obj	byte	obj	byte	obj	byte
128/4	0.1	0.9	19.1	23.5	34.4	43.0	98.6	95.0
256/20	0.4	2.2	37.1	44.2	40.0	50.3	98.3	95.3

Table 1: Fraction of globals objects in pBOB

heap/ threads	Base			TL full GCs			TL local GCs			DG global GCs			DG local GCs		
	mrk	swp	tot	mrk	swp	tot	mrk	swp	tot	mrk	swp	tot	mrk	swp	tot
128/4	327	270	597	254	348	603	134	853	987	121	149	270	221	92	313
128/12	839	514	1352	763	753	1516	145	876	1021	331	265	596	165	131	296
128/20	1947	1008	2955	1941	1685	3626	164	898	1062	619	415	1034	86	106	192
256/4	191	230	421	148	309	458	147	993	1140	48	95	142	154	70	224
256/12	362	301	663	372	512	884	168	1155	1328	89	109	198	99	84	183
256/20	557	390	947	533	609	1143	158	951	1109	129	125	254	91	89	180
256/40	1662	897	2559	1437	1300	2737	156	788	944	457	331	787	124	140	264

Table 3: Garbage collections times (in ms): details (pBOB)

heap/ threads	Cost of Write barrier
128/4	2.5%
128/12	2.7%
128/20	2.6%
256/4	2.2%
256/12	2.6%
256/20	2.1%
256/40	2.8%

Table 5: Write Barrier cost (pBOB)

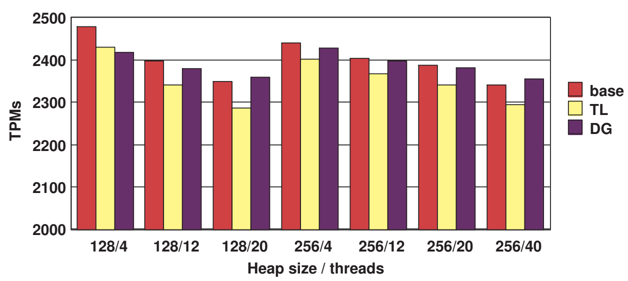

Figure 2: Throughput (pBOB)

write barrier prevents an improvement in the overall program throughput.

Figure 2 provides throughput measurements for the original JVM, the thread-local memory manager and direct global allocation. Throughput is measured in units of transactions per minute (tpm) as reported by pBOB. It turns out that throughput is reduced by around 2% when using thread-local heaps without direct global allocation. However, the throughput of the JVM with thread-local heaps and direct global allocation is very close to the original JVM. In general the improvement in pause times does not cause a noticeable loss in throughput, and there are performance gains in scenarios with high amounts of collection activity.

Let us now investigate direct global allocation. Recall that we select the program sites we use for allocating directly to the global areas. Since some sites allocate both global and local objects there is a trade-off. The more global objects

direct allocation	tpm	number full GCs	GC time
50%	-1.2%	0%	-9.4%
70%	-1.3%	0%	-15.8%
90%	-0.2%	-50%	-50.9%
99%	0.4%	-75%	-77.7%

Table 6: Improvement of direct global allocation over thread-local heaps as a function of percent global objects directly allocated as global (pBOB - 20 thread 256m heap)

we allocate directly to the global areas the more local objects get allocated to the global areas as well. Table 6 shows us that it is not useful to go part way with global allocation. The full benefits are obtained only when nearly all of the global objects are identified during allocation. The table presents the improvement of throughput, latency and garbage collection times obtained by direct global allocation over the normal thread-local heap memory manager. The improvement is presented as a function of the number of global objects that get allocated directly to the global areas. When 50% of the global objects are directly allocated as global, we see a deterioration in the throughput and almost no improvement otherwise. With 90% direct global allocation, we see GC times and the number of long pauses cut in half. Another cut in half is obtained when we increase the percentage to 99%, which is the percentage we have used in our algorithm and measurements.

Finally, we provide some measurements on how we selected the area size. See Table 7.

The collector which employed an area size of 512kb performed consistently well on both pBOB and Trade2 on WebSphere.

7.2.2 *Partial measurements for Trade2 on WebSphere driven by AKStress*

In this section we report measurements for Trade2 on WebSphere driven by AKStress. As explained above, we did not implement direct global allocation for this benchmark. Therefore, we report measurements with thread-local heaps and in the next section infer how direct global allocation would affect this case.

heap/ threads	Full GCs						Local GCs			
	number			avg gc time (ms)			number		avg gc time (ms)	
	base	TL	DG	base	TL	DG	TL	DG	TL	DG
128/4	8	6	3	74.6	100.4	90.1	315	910	12.5	1.4
128/12	10	9	4	135.3	168.4	149.0	724	963	5.6	1.2
128/20	15.4	15	5	191.9	241.7	206.8	1062	821	4.0	0.9
256/4	4	3	1	105.4	152.6	142.4	230	627	19.8	1.4
256/12	4	4	1	165.7	220.9	198.0	597	567	8.9	1.3
256/20	4.2	4	1	225.6	285.7	253.6	731	534	6.1	1.3
256/40	7	6	2	365.6	456.2	393.7	964	840	3.9	1.3

Table 4: Latency measurements (pBOB)

heap/ threads	Area size (mb)	TPMs	full GCs	local GCs	GC time (ms)
128/4	1/2	2422	3	898.2	563.3
	1	2442	3	697.8	460.3
	2	2431	5.2	329.8	599.4
128/20	1/2	2396	5	811.6	1229.6
	1	2398	6	736.4	1400.1
	2	2383	8	281.8	1752.6
265/4	1/2	2453	1	636.2	365.5
	1	2457	1.2	556.8	332.5
	2	2459	2.8	324.6	489.5
256/20	1/2	2375	1	531.6	439.1
	1	2371	2	858.2	753.8
	2	2366	3	323.0	858.0

Table 7: Choosing area size (pBOB)

heap size	calculated gc time	
	base	TL
192	34028	38460
256	23252	31874
512	12028	27062

Table 9: Garbage collections times for WebSphere (Trade2) - summary

heap size	Cost of Write barrier
192	2.1%
256	2.2%
512	1.5%

Table 12: Write Barrier cost

We start again (in Table 8) by measuring the fraction of global objects in the heap. The behavior seen in pBOB shows up here in a stronger manner. Almost no local objects survive a collection.

Next, we look at the collection times while compensating, as for pBOB for the fact that local collections are run on one processor and not 4 as the global collection. The summary of the measurements is given in Table 9 and the detailed information appears in Table 10. The pattern seen in pBOB is echoed here. The introduction of thread-local heaps increases both full collection time and total collection time. Similarly to pBOB this has a negative effect on overall throughput, which is shown in Table 3. The overhead of the write barrier, which again is around 2%, also contributes to the decline in performance (see Table 12). However, in contrast to pBOB, even without direct global allocation we obtain a clear benefit in terms of reducing the number of long pauses, as seen in Table 11.

7.2.3 *Estimating the effect of direct global allocation*

We did not implement direct global allocation for WebSphere and therefore can only estimate what its effect may be. We first recall the effect of direct global allocation for pBOB. The greatest percent gain from direct global allocation is in the total time used by local sweep. The local sweep time dropped by more than an order of magnitude, decreasing to just 7-18% of its former time. As a percentage of the local mark time, the local sweep time dropped from being 5 to 7 times greater than the local mark time to being on average less than the local mark time. Another big gain was in the global gc time. However, this is a reflection of the drop in the need for global gc. The number of global collections fell between 50-75% due to direct global allocation. We note also that the sweep time for full GC is on average less than the mark time for the original and direct global versions, but greater on the thread-local version.

We now consider some differences between the gc behavior of the two applications with thread-local collections. One difference is that for Trade2 on WebSphere, unlike pBOB, thread-local heaps alone cause the number of full gcs to drop sharply. Another difference is that the sweep time for local gc is about four times the mark time rather than five to seven times as great.

We recall two large benefits of direct global allocation. One is the need for gc is reduced because the collections recover more of the heap. This effect is due to the great reduction in fragmentation of the free space. The second benefit is that collections are more efficient in terms of the time they take to sweep (we discussed the reason for this in the previous section). For the sake of caution we estimate the effect from the second benefit only. We estimate that the least benefit from direct global allocation would be that local sweep times are reduced to the local mark times. This would give total gc time of approximately 21, 17 and 12 second for the 192, 256, and 512 megabyte heap sizes respectively. The effect would be that the overall total gc time would be reduced to as little as 62% of the original. The reduction in gc time amounts to approximately 2% of total execution time. This is enough to offset the effect degradation in execution and bring about a net gain of around 1% in the cases of the 192 and 256 heap sizes.

heap size	% surviving locals		% surviving globals		% global before gc		% global after gc	
	obj	byte	obj	byte	obj	byte	obj	byte
256	0.0	0.0	63.9	70.7	34.8	30.2	99.9	99.9

Table 8: Fraction of globals objects in WebSphere running Trade2

heap size	Base			TL full GCs			TL local GCs		
	mrk	swp	tot	mrk	swp	tot	mrk	swp	tot
192	22119	11909	34028	4373	3737	8110	6916	23435	30351
256	14212	9039	23252	2896	2519	5414	7103	20607	27710
512	6545	5484	12028	1338	1891	3229	4539	19295	23884

Table 10: Garbage collections times for Websphere (Trade2)

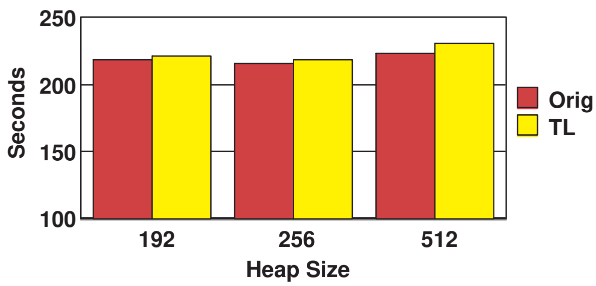

Figure 3: Overall performance - Trade2

8. CONCLUSIONS

We have presented a memory manager for local heaps, and a method of dynamically monitoring objects that are thread-local. We have implemented the algorithm for IBM's 1.3.0 Java Virtual Machine and ran measurements on an IBM Netfinity server with 550Mhz Intel Pentium III Xeon and 2GB memory. The overall garbage collection time was cut on average by about 50%. The number of long pauses decreased dramatically, by a factor of 3 to 4. The short pauses introduced are on the order of 1 ms and hence unnoticeable to the user. The improved latency did not cost in terms of throughput, which was not noticeably affected. The improvement in collection was offset by the cost of the write barrier. We believe it would be worthwhile to check how well static determination of object locality works with our memory manager.

9. ACKNOWLEDGMENTS

The authors would like to thank our colleagues in the gc groups at the Haifa Research Lab and the Java Technology Center (JTC) in Hursley for their advice and support. We would also like to thank in particular Martin Trotter of the JTC for working with us on our initial thread-local garbage collection scheme and Naama Kraus in Haifa for doing an initial feasibility study.

10. REFERENCES

[1] Blackburn, S. M., Singhai, S., Hertz, M., McKinley, K. S., and Moss, J. E. B. Pretenuring for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2001.

[2] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In OOPSLA [13], pages 35–46.

[3] Bruno Blanchet. Escape analysis for object oriented languages. application to Java. In OOPSLA [13], pages 20–34.

[4] The BulletTrain™. See http://www.naturalbridge.com/.

[5] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *1998 SIGPLAN Conference on Programming Language Design and Implementation*, pages 162–173, 1998.

[6] Jong-Deok Choi, M. Gupta, Maurice Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In OOPSLA [13], pages 1–19.

[7] R. Dimpsey, R. Arora and K. Kuiper. Java server performance: A case study of building efficient, scalable Jvms. in *IBM System Journal*, 39(1), 2000. pp. 151–174.

[8] Damien Doligez, Georges Gonthier, Portable Unobtrusive Garbage Collection for Multi-Processor Systems, *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, January, 1994.

[9] Damien Doligez, Xavier Leroy, A concurrent generational garbage collector for a multithreaded implementation of ML, *Proc. 20th Symp. Principles of Programming Languages*, 1993, pp. 113-123.

[10] T. Harris. Dynamic adaptive pre-tenuring. In Proceedings of the *International Symposium on Memory Management*, Oct. 2000.

[11] JOVE™ Optimizing Native Compiler for Java™ Technology. Technical Report. Available at http://www.instantiations.com/jove/jovereport.htm.

[12] F. Lockwood Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–5, 1978.

[13] *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, Denver, CO, October 1999. ACM Press.

[14] Standard Performance Evaluation Corporation, `http://www.spec.org/`.

[15] Bjarne Steensgaard. Thread-Specific Heaps for Multi-Threaded Programs. In the *2000 International Symposium on Memory Management* October, 2000.

[16] Trade2. `http://www.ibm.com/servers/eserver/pseries`

heap size	Full GCs				Local GCs	
	number		avg gc time		number	avg gc times
	base	TL	base	TL	TL	TL
192	131	26	260.7	311.9	53427	2.3
256	84	18	276.8	309.4	40955	2.6
512	36	7	338.8	461.3	21316	4.5

Table 11: Pause frequency and time for WebSphere (Trade2)

/hardware/whitepapers/websphere_m80.html

[17] WebSphere. http://www.ibm.com/websphere/.

[18] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In OOPSLA [13], pages 187–206.