Packer: an Innovative Space-Time-Efficient Parallel Garbage Collection Algorithm Based on Virtual Spaces

Shaoshan Liu[1], Ligang Wang[2], Xiao-Feng Li[2], and Jean-Luc Gaudiot[1]

[1] EECS, University of California, Irvine

[2] Intel China Research Center

ABSTRACT— *The fundamental challenge of garbage collector (GC) design is to maximize the recycled space with minimal time overhead. For efficient memory management, in many GC designs the heap is divided into large object space (LOS) and non-large object space (non-LOS). When one of the spaces is full, garbage collection is triggered even though the other space may still have a lot of free room, thus leading to inefficient space utilization. Also, space partitioning in existing GC designs implies different GC algorithms for different spaces. This not only prolongs the pause time of garbage collection, but also makes collection not efficient on multiple spaces. To address these problems, we propose Packer, a space-and-time-efficient parallel garbage collection algorithm based on the novel concept of virtual spaces. Instead of physically dividing the heap into multiple spaces, Packer manages multiple virtual spaces in one physically shared space. With multiple virtual spaces, Packer offers the advantage of efficient memory management. At the same time, with one physically shared space, Packer avoids the problem of inefficient space utilization. To reduce the garbage collection pause time of Packer, we also propose a novel parallelization method that is applicable to multiple virtual spaces. We reduce the compacting GC parallelization problem into a tree traversal parallelization problem, and apply it to both normal and large object compaction.*

1. INTRODUCTION

Garbage collection technology has been widely used in managed runtime systems, such as Java virtual machine (JVM) and Common Language Runtime (CLR) systems. For efficient memory management, a modern high performance GC usually manages large and normal objects separately such that the heap is divided into large object space (LOS) and non-large object space (non-LOS). However, the object size distribution varies from one application to another and from one execution phase to the next even in one application, thus it is impossible to predefine a proper heap partitioning for LOS and non-LOS. Existing GCs with separate allocation spaces mostly suffer from the problem that they do not fit well with the dynamic variations of object size distribution at runtime. This problem leads to imbalanced space utilization and thus negatively impacts the overall GC performance. For garbage collection algorithms, conventional mark-sweep and reference counting collectors are susceptible to fragmentation. To address this problem, copying or compacting GCs are introduced. Compaction eliminates fragmentation in place by grouping live objects together in the heap and freeing up large contiguous spaces for future allocation. As multi-core architectures prevail, parallel compaction algorithms have been designed to achieve time efficiency. However, large object compaction is hard to parallelize due to strong data dependencies such that the source object can not be moved to its target location until the object originally in the target location has been moved out. Especially, the parallelism is seemingly inadequate when there are few large objects.

In this paper, we propose Packer, a space-time-efficient parallel garbage collection algorithm based on the novel concept of virtual spaces. Unlike some conventional garbage collectors [2] that physically divide the heap into multiple spaces, Packer manages multiple virtual spaces in one physical space. With multiple virtual spaces, Packer offers the advantage of efficient memory management, so that different virtual spaces can employ best suitable collection algorithms. Meanwhile, with one physical space, Packer avoids the problem of inefficient space utilization, since there is no space partitioning problem any more. Object allocation is highly efficient in Packer. The free space in the physical heap is centrally controlled by a tree structure. When one of the virtual spaces needs more space, then it searches the tree to fetch a suitable free region. In particular, normal object allocation is done in thread local blocks with bump-pointers, requiring no synchronization. Garbage collection is triggered only when the heap contains no free region, thus guaranteeing that the heap is fully utilized. Packer supports both compaction and mark-sweep for large objects. Hence, it incorporates the advantages of both the Mark-Sweep and Compaction algorithms, and is able to achieve high performance when either algorithm is suitable. To further reduce the garbage collection pause time of Packer, we reduce the compacting GC parallelization problem into a tree traversal parallelization problem, and apply it to both normal and large object compaction.

In this paper we present the design details of the proposed algorithms and evaluate their efficiencies. These algorithms are implemented in Apache Harmony, a product-quality open source JAVA SE implementation. The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 introduces the basic algorithm designs of Packer. Section 4 presents the parallelization of normal and large object compaction in Packer. Section 5 presents the evaluation results with Specjbb2005 and Dacapo benchmark suites. Finally, section 6 summarizes the project and discusses future work.

2. RELATED WORK

Conventional garbage collectors utilize Mark-Sweep algorithms [14, 15, 16] to manage the whole heap. When the heap has no more free space, GC is triggered and the collectors start to trace the heap. If an object can be reached, then it is a live object and the object header is marked. After tracing, all unmarked objects are swept and their occupied spaces are recycled. The free space in the heap is managed with a linked list. During allocation, the allocators traverse this linked list to fetch a suitable free region. The main advantage of this algorithm is that no data movement is necessary, such that it incurs low overhead during garbage collection. However, it also has three disadvantages: first, this algorithm introduces heavy fragmentation on the heap, leading to inefficient space utilization. Second, its fragmentation property destroys the spatial locality of data allocation, leading to inefficient data access. Last, each data allocation requires a linear search on the linked list that manages the free regions, thus it is inefficient.

As exemplified by the LISP2 algorithm [5], compaction algorithms are utilized in GC designs to address the disadvantages of mark-sweep algorithms. However, compaction usually imposes lengthy pause time. To reduce pause time, several parallel compaction algorithms have been proposed. Flood *et al*. [6] present a parallel compaction algorithm that runs three passes over the heap. First, it determines a new location for each object and installs a forwarding pointer, second it fixes all pointers in the heap to point to the new locations, and finally, it moves all objects. To make this algorithm run in parallel, the heap is split into N areas such that N threads are used to compact the heap into N/2 chunks of live objects. The main disadvantage of this design is that the resulted free space is noncontiguous. Abuaiadh *et al.* [7] propose a three-phase parallel compactor that uses a block-offset array and mark-bit table to record the live objects moving distance in blocks. Kermany and Petrank [8] propose the Compressor that requires two phases to compact the heap; also Wegiel and Krintz [9] design the Mapping Collector with nearly one phase. Both approaches depend on the virtual memory support from the underlying operating system. Note that concurrent GC and Stop-The-World (STW) GC designs are fundamentally different: they have different design goals, evaluation metrics, and algorithms: concurrent GC is designed to reduce pause time, whereas STW GC is designed to increase throughput. In this paper, we only focus on STW GC design.

For efficient memory management, Caudill and Wirfs-Brock first propose to use separate spaces to manage objects of different sizes, large object space (LOS) for large objects and non-large object space (non-LOS) for normal objects [1]. Hicks *et al*. have done a thorough study on large object spaces [2]. The results of this study indicate three problems for LOS designs. First, LOS collection is hard to parallelize. Second, LOS shares the same heap with non-LOS, thus it is hard to achieve full utilization of the heap space. Third, LOS and non-LOS collections are done in different phases, which may affect the scalability of parallel garbage collection. In [3], Soman *et al*. discuss about applying different GC algorithms in the same heap space, but their work does not involve dynamically adjusting the heap partitioning. The study done by Barrett and Zorn [4] is the only known publication that studies space boundary adjustment, and their work aims at meeting the resource constraints such as pause time. By contrast, Packer does not require any boundary adjustment mechanism. Instead, it manages different virtual spaces in the same physical space such that it avoids the problem of inefficient space utilization while keeping the advantage of efficient memory management.

3. BASIC ALGORITHM DESIGNS IN PACKER

In this section, we first introduce the basic heap design of Packer and compare it to other heap designs. Then we present the data allocation scheme and garbage collection algorithm in Packer. At last, we discuss further implications of the Packer design.

3.1 The Basic Design of Packer

As shown in Figure 1, with the Move-Compact algorithm (GC-MC), when the heap is partitioned into multiple spaces, for instance LOS and non-LOS, garbage collection is triggered when either space is full. In times when garbage collection is triggered by one space while the other space is partially filled, the heap is not fully utilized. Consequently, it leads to more frequent collections and lower performance.

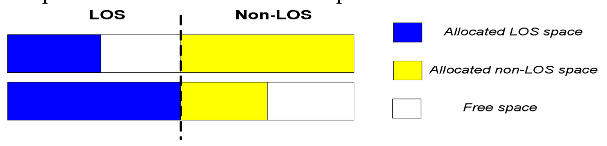

Figure 1: Compacting GC (GC-MC) with separate allocation spaces

The key question here is why one space would get full before the other one does. This is because within the same amount of time, a higher fraction of one space's free region is allocated than that of the other space, yet the free regions of different spaces can not be easily shared. Hence, if the free regions of the heap are centrally controlled and can be shared by both spaces, then the problem of low space utilization can be solved. However, this is not possible when a heap is partitioned into two spaces with a boundary in between. Because it requires the GC algorithm for one space virtually manage the regions of another space when its own space is full. For instance, as shown in the lower part of Figure 1, if LOS is full, we cannot allocate large objects in the free region of non-LOS. Otherwise, we destroy the advantage brought by separate spaces, namely, efficient memory management.

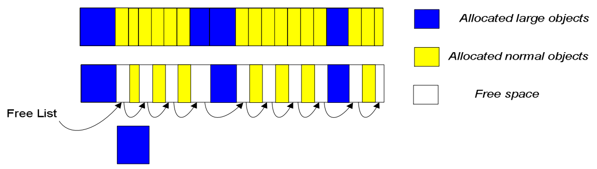

Figure 2: Mark-Sweep GC (GC-MS)

On the other hand, Mark-Sweep algorithm (GC-MS) does not divide the heap into separate spaces thus both normal and large objects share the same allocation space. When garbage collection is triggered, it scans the heap and marks all live objects. Then it sweeps all the unmarked objects, leaving holes in the heap. These holes are then added into a linked list for future allocations. Although this algorithm is efficient and does not require object movement, it introduces several serious problems. As shown in the lower part of Figure 2, in cases when fragmentation is very serious, it is unable to find a hole on the linked list to fit a newly allocated object, such as the one at the bottom of Figure 2, even though the heap has about 50% of free space. This creates a "deadlock" situation, in which compaction has to be initiated in order to alleviate the problem. In addition, GC-MS's allocation scheme breaks the spatial locality: when it allocates several objects that are meant to be accessed continuously, it has to allocate these objects sparsely all over the heap. Furthermore, for the allocation of each object, it has to traverse the free-region linked list to find a suitable free region, which is inefficient, especially when the list is long.

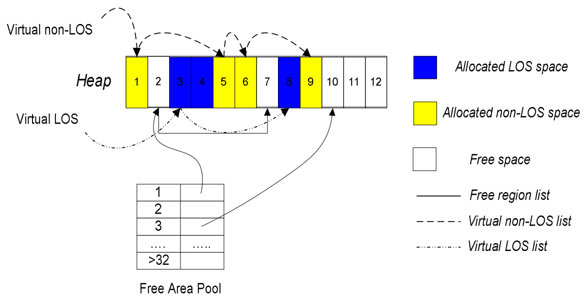

Figure 3: the design of heap structure in Packer

Packer is able to solve these problems by managing multiple virtual spaces in one physical space, such that these virtual spaces can share the free regions. As shown in Figure 3, to coordinate data management in Packer, we utilize three data structures: a virtual non-LOS list, a virtual LOS list, and a Free Area Pool. The virtual non-LOS list points to the first normal object block in the heap, and this block contains a pointer that points to the next normal object block, and so on. Hence, it is easy to find all normal blocks through this virtual non-LOS list, and they form the virtual non-Large Object Space. Similarly, the virtual LOS list points to the first large object, and this large object contains a pointer to the next large object, and so on. The virtual LOS list and the blocks of large objects form the Large Object Space. The Free Area Pool manages all free blocks in the heap, and it is actually a table of linked lists indexed by the number of blocks. Each linked list in the Free Area Pool manages all free regions with a certain number of contiguous blocks. For instance, block 2 and block 7 in Figure 3 are both free regions with only one block. Hence slot 1 of the Free Area Pool contains a pointer to block 2, and block 2 contains a pointer to block 7. Also, block 10, 11, and 12 form a free region that contains 3 free blocks, and thus slot 3 of the Free Area Pool contains a pointer to block 10. For all free regions that contains more than 32 free blocks, Packer organizes them in slot >32. In this way, all free blocks in the heap are centrally managed. With this design, the virtual spaces can grow based on need and garbage collection only happens when the heap is fully utilized.

It is relatively easy to come up with a design that manages the heap with block-number indexed table, the difficulty and subtlety in Packer actually lie in its allocation and collection algorithms. The first glimpse of Packer heap structure might give an impression that Packer is similar to a mark-sweep GC that employs size-segregated lists as the main data structure, but different from a mark-sweep GC, Packer can achieve the benefits of fast bump-pointer allocation, and parallel compacting collection for both large and normal objects, which are not possible in a mark-sweep GC. In following subsections, we will describe the object allocation and garbage collection mechanism in Packer.

3.2 Object Allocation in Packer

When multiple threads are running in an application program, they share the heap resources thus accesses to the Free Area Pool for object allocation need to be synchronized. However, if one atomic operation is required for each object allocation, then the overhead would be very serious. In most application programs, the majority of objects are normal objects which are much smaller than the block size. Thus it is essential to have an efficient allocation scheme for normal objects. The Mutator thread is responsible for object allocation. To reduce the synchronization overhead, in Packer, each Mutator fetches a thread local block from the Free Area Pool through an atomic operation, and then allocates normal objects on this thread local block with bump-pointer allocation. Note that when objects are allocated locally by the Mutator, no synchronization operation is necessary. When there is no more space in this thread local block, the Mutator atomically fetches another thread local block from the Free Area Pool and starts allocation again. In this way, only one atomic operation is required for each block instead of for each object.

To guarantee fast normal object allocations, Packer only allocates thread local block from slot 1 or slot >32 in the Free Area Pool. It first checks if slot 1 is null, if not, it allocates from slot 1; otherwise it allocates from the last slot, slot >32. In this way, it only requires one atomic operation for each thread local block. To grab a region from other slots, Packer needs to pick off the region, allocate a block, and put back the rest into the corresponding slot, which requires two atomic operations. When picking the thread local block from slot 1, one atomic operation is enough because it never needs to put back the rest. For thread local block allocation in slot >32, instead of picking off a region, Packer simply reduces the number of blocks of a region in the last slot. This reduction operation is atomic thus it guarantees thread-safe block allocation and only one atomic operation is needed.

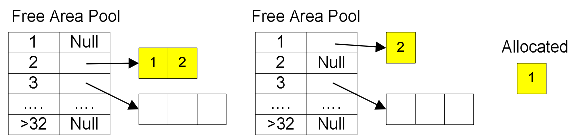

Figure 4: block allocation from the Free Area Pool

Different from normal objects, each large object occupies one or more blocks. Thus, the Mutators directly allocate large objects in the Free Area Pool. When there is an allocation request, a Mutator first checks the number of blocks requested, *n*. Then it searches the Free Area Pool using *n* as index. If slot *n* is Null, then it searches slot *n*+1 and so forth until a non-Null slot is found. Next, it fetches the first free region from this non-Null slot and grabs *n* blocks, then storing the rest back into the Free Area Pool. As an illustration in Figure 4, it requests one free block but slot 1 of the Free Area Pool is Null. Then it searches down and fetches a free region in slot 2. This free region contains free blocks 1 and 2. Packer allocates block 1 and stores block 2 back into the Free Area Pool, and it is inserted into slot 1, since now there is only one free block, block 2, left in this free region. With this design, Packer can achieve fast object allocation for both the large and normal objects, much faster than a mark-sweep GC. Next we describe how Packer achieves fast garbage collection.

3.3 Garbage Collection in Packer

When the heap is fully occupied by normal and large objects, garbage collection is triggered. Packer utilizes compaction algorithms, and its garbage collection is divided into four phases. In the first phase, it scans the heap and marks all live objects, then it builds the virtual spaces by adding all normal blocks into the virtual non-LOS linked list, and all large blocks into the virtual LOS linked list. This phase corresponds to lines 1 and 2 in the *Packer_Compact_Collection()* pseudo-code shown in Figure 5. In phase 2, normal blocks are compacted towards the left of the heap and the forwarding tables are set in each block. These forwarding tables store the offsets between the source and target addresses of objects, and they are used for the reference fixing operation in the next phase. This phase corresponds to lines 3, 4, and 5 in the pseudo-code. In phase 3, Packer fixes all the references from both normal and large objects using the forwarding tables set in the previous phase. In this case, if there is a reference pointing to an object that has already been compacted, it checks the forwarding table in this block to look for the address offset. Then it subtracts this offset from the original address stored in the reference to get the new address of this object. This phase corresponds to line 6 of the pseudo-code. In the last phase, large blocks are compacted and the free blocks are added to the Free Area Pool. This phase corresponds to line 7, 8, and 9 in the pseudo-code.

```
Procedure Packer_Compact_Collection()

Begin

1.  parallel_mark_scan_heap();
2.  build_virtual_spaces();
3.  parallel_move_normal_objects();
4.  if(need_move_large_object)
5.    compute_large_object_target_address();
6.  parallel_fix_object_references();
7.  if(need_move_large_object)
8.    parallel_move_large_object_block();
9.  add_free_area_into_free_area_pool();

End
```

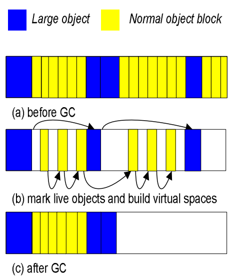

Figure 5: Garbage Collection algorithm in Packer

Packer can optionally choose not to compact large objects, such that large objects are mark-swept. With this support, Packer incorporates the advantages of both GC-MS and GC-MC. For the two extreme cases: 1) if it is large-object-intensive, Packer can choose to mark-sweep large objects, as GC-MS does, thus avoiding the object moving overhead; 2) if there are few large objects in the application, Packer behaves the same as GC-MC, thus creating a large contiguous free region while keeping the object moving overhead low. Note that the major steps, marking, normal object moving, reference fixing, and large object moving, which correspond to steps 1, 3, 6, and 8 in pseudo-code, are fully parallelized. The detailed parallelization mechanism will be explained in the next section.

Since all the live objects are identified during the first marking phase, the compaction algorithm can pack all the live objects to one end of the heap without any holes unfilled. There is no fragmentation issue. Also note that Packer compacts the normal objects before the large objects: it squeezes out large contiguous free space after the normal object compaction and then uses the free space for large object compaction.

3.4 Further Implications of Packer

Besides the advantages in object allocation and garbage collection, the Packer design has three further implications: it facilitates pinned object management, the management of managed and native data in the same heap, and the management of discrete physical areas. Pinned object support is required in some runtime systems that use conservative GC. When a garbage collector scans the heap for live objects, sometimes it will trace to a location, the content of which (pointer or value) is unknown. In this situation, conservative garbage collectors would assume that it stores an address to an object. However, the collector cannot update this reference slot because it may be storing a value instead of an address. Thus, this object is a pinned object because it cannot be moved. In conventional moving GC designs, the algorithms have to sacrifice performance when there is a pinned object in the heap. By introducing the concept of virtual spaces, Packer is able to jump over the block containing pinned objects when building the virtual spaces without any performance compromise.

Pinned object is also desirable if there are lots of interactions between managed code and unmanaged code in the application. In most Virtual Machines, including JVM and CLR, the managed and unmanaged environments often need to communicate with each other and pass data around. The most common approach to deal with this problem is to copy data from the managed environment to the native environment and vice versa [17]. This is highly inefficient because large amount of data copying incurs very high time and space overheads. Indeed, this problem can be solved by either temporarily disabling garbage collection or pinning the data passed across the boundary. With Packer's support of pinned object, this problem can be easily resolved.

Also, in some cases, a process's address space is segmented by the operating system. For example, the system may load DLLs to arbitrary address ranges, thus breaking the heap into multiple chunks. Packer is able to link these discrete chunks to create a virtual heap for the process, therefore providing an as large as possible managed heap to the applications, making the heap management efficient.

4. PARALLELIZATION OF GARBAGE COLLECTION IN PACKER

In this section we first demonstrate how we reduce the compaction parallelization problem into a tree traversal parallelization problem. Then we present the implementation of parallel normal and large object compaction in Packer.

4.1 Parallelization of Compacting GC

Compacting GCs move live objects towards one end so as to eliminate fragmentations. In order to increase GC efficiency, parallel compaction algorithms are essential in modern GC designs. The fundamental goal of a parallel compaction algorithm is to exploit as much parallelism as possible while keeping the synchronization overhead as low as possible.

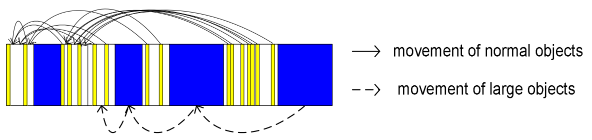

Figure 6: normal and large object compaction

As shown in Figure 6, there exist many normal objects in virtual non-LOS, and the data dependencies between these normal objects are fairly low, implying a high degree of parallelism. However, in order to parallelize the compaction process in a straightforward manner, an atomic operation, which is notorious for its inefficiency, is needed for each object movement. Thus the cost of parallelization may well surpass the performance gain. On the other hand, there exist strong data dependencies in virtual LOS such that the source object can not be moved to its target location until the object originally in the target location has been moved out. Especially, when there are only few very large objects, the parallelism is seemingly inadequate.

This observation indicates that we need to set a proper parallelization granularity in the heap such that it reduces the high synchronization overheads caused by fine-grain data movement (as in virtual non-LOS) and the false data dependencies caused by coarse-grain data movement (as in virtual LOS). Our design is to divide the heap into equal-sized blocks. This means that the parallelization granularity is a block. For virtual non-LOS, each block contains multiple objects. During collection, each thread obtains a block and moves all the objects in the block. Thus, at most one atomic operation is required for the movement of multiple objects in a block, greatly reducing the synchronization overhead. On the other hand, for virtual LOS, each object contains one or more blocks. When one block of a large object can not be moved due to data dependency, the other blocks can still be moved, thus reducing the false dependency problem. For instance, in Figure 7, blocks 7 and 8 belong to one object, and blocks 11 and 12 belong to another. Originally, the blocks of one large object must be moved together, so blocks 7 and 8 cannot be moved until block 5 has been moved out. With equal-sized blocks, dependencies only exists between block 7 and 5, so the dependencies between block 8 and 5 are false data dependencies. Block 8 can be moved independently of block 7. Thus the movements of block 7 and 8 can be parallelized.

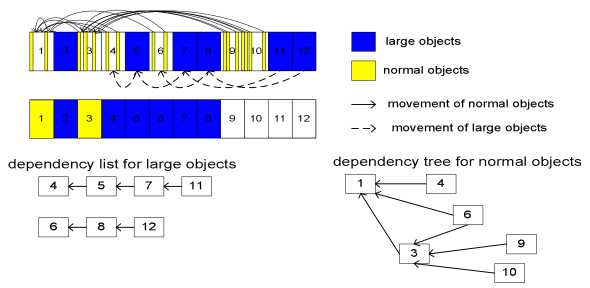
1
2
3
4
5
6
7
8
9
10
11
12
large objects
normal objects
movement of normal objects
movement of large objects
dependency list for large objects
dependency tree for normal objects

Figure 7: Packer with equal-sized blocks

By dividing the heap into equal-sized blocks, both false data dependencies and synchronization costs can be reduced. To capture all data dependencies and to facilitate the parallelization of data compaction, dependency lists are generated for virtual LOS and a dependency tree is generated for virtual non-LOS.

Further complications exist in parallelizing the compaction process. For virtual non-LOS, races between multiple collectors exist when they move objects from a source block to a target block. For instance, two collectors may move data from two source blocks into the same target block, or one collector may write into a target block in which the original objects have not been moved away yet. This observation indicates two properties. First, each block has two roles, it is a source block when its objects are compacted to some other block, and it can be a target block after its original data has been moved away. Second, in virtual non-LOS, multiple source blocks may compact into one target block, and thus the access to this target block should be synchronized. In order to achieve high performance, the complex relations between the blocks need to be clarified before the compacting threads start. To achieve this, we generate dependence trees, such as the one in Figure 7, which captures all the data dependencies between the blocks. For instance, in virtual LOS, block 5 is the source block for block 4 and it is also the target block for block 7. Thus, block 7 cannot be moved to block 5 until block 5 has been moved to block 4. In virtual non-LOS, block 1 is the target block for block 3, and block 3 is also the target block for blocks 6, 9 and 10. Thus, blocks 6, 9, and 10 cannot be moved to block 3 until block 3 has been moved to block 1. When compaction starts, the threads traverse the tree to obtain a source block and a target block. After the current data movement is done, the thread moves down the tree to obtain a new source block and set the old source block to be the new target block. This process finishes after the thread has reached the leaf nodes of the tree. We have thus actually reduced the compaction parallelization problem into a tree traversal parallelization problem. For virtual LOS compaction, the situation is simpler because one source block has only one target block, and vice versa. Therefore, the dependency trees degenerate into dependence lists.

4.2 Implementation of Parallel Normal Object Compaction

Packer utilizes the Move-Compact algorithm from Apache Harmony JVM for normal object compaction [7]. This algorithm involves three phases for parallel normal object compaction: live object marking, object moving, and reference fixing.

<u>Phase 1</u>: *Live object marking.* It traces the heap from root set and marks all the live objects;
<u>Phase 2</u>: *Object moving.* It copies the live objects to their new locations;
<u>Phase 3</u>: *Reference fixing.* It adjusts all the reference values in the live objects to point to the referenced objects' new locations.

Although the three phases are fully parallel, we only focus on the parallelization of the moving phase, which is most related to our proposed design. In this phase, a collector first atomically grabs a source block in heap address order. Then it grabs a target block that has lower address than the source block. Each block is divided into multiple sectors that each encapsulates a number of live objects. For each sector of live objects in the source block, the collector computes its target address in the target block, moves the sector to its target position, and stores the address offset to the forwarding table in the block header. When the target block has not enough space, the collector grabs the next target block. When the source block has no more live objects, the collector grabs another source block in heap address order until all the blocks have been visited. In this phase, two atomic operations are needed for one block to eliminate data races: one for taking the ownership of the source block, and the other for taking the ownership of the target block. Note that this process can be seen as a parallel tree traversal process. When a collector grabs a source block and a target block in heap address order, it is actually traversing from the top of the tree. When it finishes the movement of data in the current source block, the source block is released and can be used as a target block in the next iteration, thus the collector is indeed traversing down the dependency tree until all blocks have been compacted. If multiple collectors try to grab the same target block, synchronization mechanism is necessary to coordinate their operations. Note that in this three-phase algorithm, target address calculation and object movement is done in the same phase, thus the dependency tree is generated dynamically instead of pre-generated.

```
Procedure Parallel_Large_Object_Compaction()
Begin
    dep_list = get_next_compact_dep_list();
    while(dep_list){
     target_block = get_first_block(dep_list);
     source_block = get_next_block(dep_list);
     while(source_block != NULL){
       memmove(target_block, source_block);
       target_block = source_block;
       source_block = get_next_block(dep_list);
      }
     dep_list = get_next_compact_dep_list();
    }
End
```

Figure 8: parallel large object compaction

4.3 Implementation of Parallel Large Object Compaction

To demonstrate the effect of the parallel virtual LOS compaction algorithm, we implemented the parallel compaction algorithm presented above in the Apache Harmony GC and the pseudo-code is shown in Figure 8. Before collection starts, a number of disjoint dependence lists are generated to capture the dependence relationship among the large object blocks. Each collector can then atomically grab a dependence list and works on it independently. In this case, it only requires an atomic operation for each dependency list instead of for each block. In essence, a thread first acquires the ownership of a dependency list through an atomic operation. From the list, it gets the first block, which is the target block, and the second block, which is the source block, and moves the source to the target. When it finishes this block movement, the source block now becomes the target block and a new source block is obtained by taking the next block in the dependency list. This operation repeats until there is no more block in the dependency list. Then, the thread obtains another dependency list from the task pool.

5. EXPERIMENTS AND RESULTS

In this section, we present our experiment results for our Packer algorithm. All proposed algorithms have been implemented in Apache Harmony, a product-quality open source JAVA Virtual Machine [10]. The heap is divided into equal-sized blocks, and each block contains a block header for its metadata, including block base address, block ceiling address, block state, etc. Block size is adjustable, but the block header size is a constant and independent of the block size. For this study, the block size is set to 32 KB and the size threshold for large objects is set to 16 KB. The evaluation of Packer is done with the SPECjbb2005 [11] and Dacapo [12] benchmark suites. SPECjbb2005 is a large server benchmark that employs several program threads; it is representative of commercial server-side applications. On the other hand, Dacapo is a suite of client-side Java applications. For all experiments, we use a 256 MB heap by default.

In these experiments, we compare three GC designs: GC-MC, GC-MS, and Packer. GC-MC is the default GC algorithm in Apache Harmony and it utilizes the Move-Compact algorithm for garbage collection. It divides the heap into separate spaces: Large Object Space (LOS) and non-LOS, to manage large and normal objects. However, this algorithm can not be parallelized for the compaction of large objects. For GC-MC, with a heap size of 256M, we experimented with four configurations: GC-MC with 50M LOS (GC-MC 50M), GC-MC with 100M LOS (GC-MC 100M), GC-MC with 150M LOS (GC-MC 150M), and GC-MC with 200M LOS (GC-MC 200M). GC-MS uses Mark-Sweep for the garbage collection of the whole heap. Packer manages virtual LOS and virtual non-LOS in the same heap, and enables the parallelization of both normal and large object compactions.

5.1 Comparison of Space Utilization

In real applications, the object size distribution varies from one application to another and from one execution phase to next even in one application. For instance, specjbb2005 is a non-large-object-intensive benchmark that allocates a very small number of large objects, thus it requires a large non-LOS. On the other hand, xalan, jython, and bloat from the Dacapo benchmark suite are large-object-intensive thus requiring a large LOS. In addition, specjbb2005 allocates all the large objects at the beginning of its execution and very few large objects afterwards. Thus in different phases of its execution, it requires different sizes for LOS.

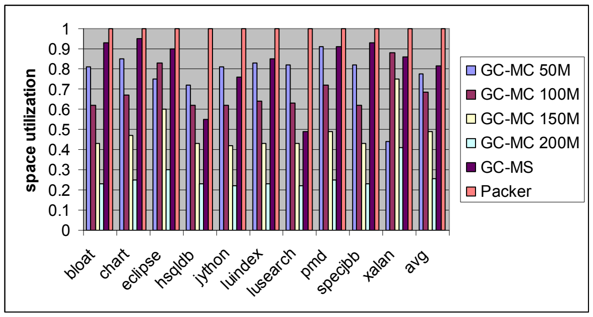

Figure 9: Space Utilization of GC-MC, GC-MS, and Packer

Y-axis shows the fraction of utilized heap space when garbage collection happens. This result is obtained by averaging the space utilization of all garbage collections throughout execution. The last set of results, avg, compares the average heap utilization of all selected benchmarks on different GC designs.

Figure 9 shows the space utilization of different designs. Packer guarantees the heap space is fully utilized because collection is triggered only when there is no free region in the Free Area Pool. The average space utilization of GC-MS is 81%. For lusearch, the space utilization is only 49%, which is caused by heavy fragmentation. The average space utilization ratios are 78%, 69%, 49%, and 26% for GC-MC 50M, GC-MC 100M, GC-MC 150M, and GC-MC 200M respectively. Usually in application programs, most objects are normal objects. Hence when LOS gets too big, there is insufficient space for normal object allocation, causing frequent garbage collections and low space utilization. Nonetheless, for Xalan, GC-MC space utilization is maximized when LOS size is 100M. This is because Xalan is a large-object-intensive application, which contains a large number of large objects when garbage collection happens. In general, space utilization is worse when the heap is statically partitioned into multiple spaces. Static partition fails to meet the needs of large object and non-large object space utilization, precisely because this is a dynamic behavior. On the other hand, although GC-MS does not suffer from this problem, it creates a heavy fragmentation problem, often leading to low space utilization. By managing multiple virtual spaces in one physical space, Packer overcomes all these problems.

Table 1: GC pause time comparison of Packer, GC-MS, and GC-MC

	Packer		**GC-MS**		**GC-MC 50M**		**GC-MC 100M**		**GC-MC 150M**		**GC-MC 200M**	
eclipse	*199*	*1863*	*206*	*1602*	*291*	*2427*	*225*	*2048*	*338*	*2670*	*845*	*5474*
hsqldb	*32*	*1410*	*F*	*F*	*44*	*2085*	*125*	*5975*	*F*	*F*	*F*	*F*
Jython	*190*	*1109*	*266*	*436*	*232*	*1259*	*303*	*1396*	*455*	*1686*	*918*	*2573*
lusearch	*84*	*3389*	*175*	*3465*	*100*	*3707*	*134*	*4323*	*203*	*5553*	*416*	*9331*
specjbb	*39*	*1204*	*41*	*1160*	*119*	*3927*	*F*	*F*	*F*	*F*	*F*	*F*

Table 1 shows the single-thread performance of Packer, GC-MS, and GC-MC, including the number of garbage collection events triggered throughout execution (left column) and the total GC pause time (right column). The first observation is that Packer always triggers fewer garbage collections compared to other designs. This is because Packer guarantees that the heap is fully utilized. The second observation is that some applications fail to finish execution, as those denoted “F” in the table. For Specjbb and hsqldb, some GC-MC configurations with large LOS size fail to complete because they do not have sufficient space for normal object allocation. In addition, for hsqldb, GC-MS fails to complete because of heavy fragmentation. This happens when it can not find suitable free region in the heap for the newly allocated object. The third observation is that GC-MS usually has lower pause time than both Packer and GC-MC. One extreme case is jython, in which GC-MS’s pause time is only 1/3 of that of Packer. This is because Mark-Sweep does not involve object movement, which may incur a high performance overhead.

5.2 Scalability of Packer

To demonstrate the effect of Packer’s parallel compaction algorithms, we compare the scalability of Packer and GC-MS with 1, 2, 3, and 4 threads and the results are shown in Figures 10 and 11. The Y-axis of these figures represents the normalized total GC pause time. Figure 10 shows Packer’s scalability. In general, Packer demonstrates very good scalability. On average, the speedups of Packer are 1.92x, 2.64x, and 2.67x respectively with 2, 3, 4 collectors.

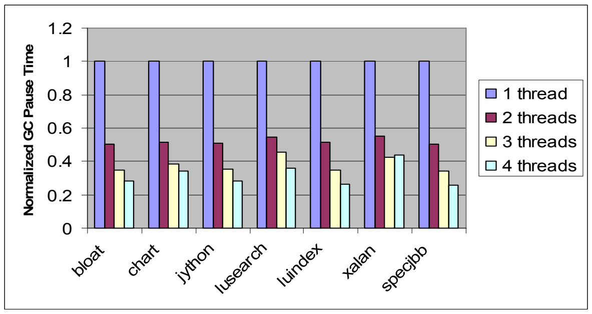

Figure 10: Packer Scalability

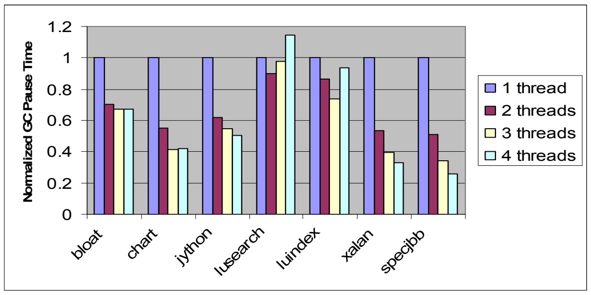

Figure 11: Mark-Sweep Scalability

Figure 11 shows GC-MS's scalability. Compared to Packer, GC-MS's scalability is lower. On average, the speedups of GC-MS are 1.5x, 1.72x, and 1.64x respectively with 2, 3, 4 collectors. Note that the average speedup for the 4-thread case is actually lower than that of the 3-thread case. This is because for some benchmarks, such as lusearch and luindex, the 4-thread case introduces long pause time. This is particularly true for lusearch, where the pause time for the 4-thread case is much higher than the sequential case due to the heavy fragmentation in these applications. When fragmentation is serious, garbage collections become much more frequent and the elapsed time between two garbage collections is very short. Hence, only a small number of objects are allocated and collected in each allocation-collection period. Under this situation, synchronization overhead becomes the major component of the GC pause time, negatively impacting GC performance. For other applications with low degree of fragmentation, such as xalan and specjbb, the speedups are comparable to those of Packer.

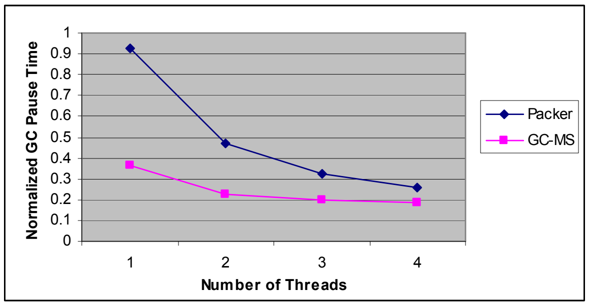

Figure 12: Comparison of Parallel Packer and GC-MS

Table 1 indicates that in the sequential case, Mark-Sweep is more efficient than compaction algorithms because it does not involve the movement of objects. Nevertheless, as the number of threads increases, Packer gradually takes the performance advantage over GC-MS due to better scalability. As an illustration, in Figure 12 we compare the performance of parallel GC-MS and Packer on jython. It clearly shows that although GC-MS's GC pause time is only 1/3 of that of Packer in the sequential case, these two numbers converge as the number of threads increase.

5.3 Impacts on Overall Performance

This section presents how Packer impacts the performance of the overall program execution. To collect this data, we run the respective benchmarks on an Intel 8-core Tulsa platform and compare the performance of Packer, GC-MC, and GC-MS. For GC-MC, we manually optimized the LOS size to maximize space and time efficiency for each application. Figure 13 shows the results on Specjbb2005 benchmark. The X-axis shows the number of warehouses used in execution and the Y-axis shows the normalized Specjbb score, a higher score represents higher performance. Packer's performance is almost always 1.2% higher than that of GC-MS. Although this seems to be a very small performance gain, but considering that garbage collection only takes about 10% of the total execution time, this result is actually a great improvement on GC performance. Also, Packer's performance is higher than that of GC-MC, but the advantage is not obvious. This is because both GC-MC and Packer utilize the same algorithm for normal object compaction and Specjbb2005 is not a large-object-intensive benchmark.

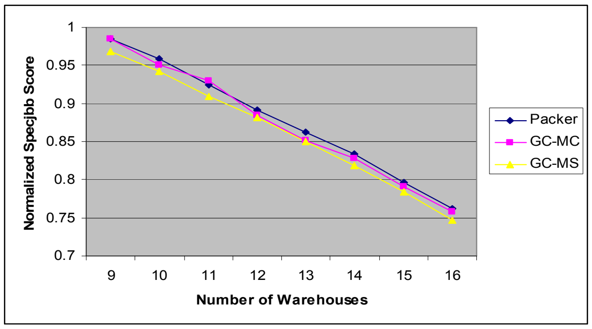

Figure 13: Impacts on Specjbb2005 Overall Performance

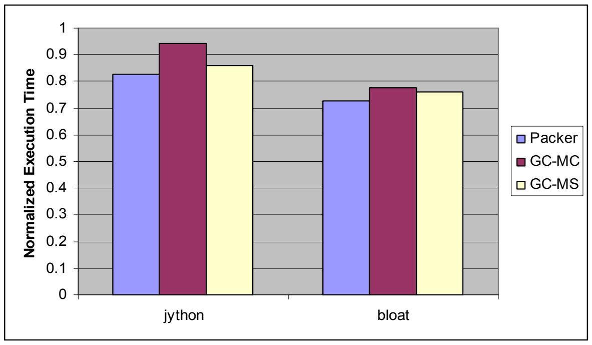

Figure 14: Impacts on Dacapo Overall Performance

Figure 14 presents the results with the Dacapo benchmark suite. Compared to Specjbb, jython and bloat are large-object-intensive. Packer's performance is 3% higher than that of GC-MS and 8% higher than that of GC-MC. Note that in GC-MC, large object compaction is not parallelized. Thus in sequential case, the Mark-Sweep algorithm has better performance than compaction in large object garbage collection. Nevertheless, with the parallel large object compaction algorithm proposed in this paper, compaction can be more efficient compared to Mark-Sweep.

6. CONCLUSION

Space and time efficiency are the two most important design goals in garbage collector design. However, many garbage collection algorithms trade space utilization for performance and vice versa. In this paper, we proposed Packer, a novel garbage collection algorithm that manages multiple virtual spaces in one physical space, thereby guaranteeing the space is fully utilized while avoiding the fragmentation problems. To improve performance, we first reduced the heap compaction parallelization problem into a parallel tree traversal problem, and then designed solutions to eliminate false sharing and to reduce the synchronization overhead, thereby maximizing the exploitable parallelism for both normal and large object compaction. It is noteworthy that Packer is generic enough to be used in any situation that involves the management and coordination of multiple virtual spaces in one physical space and vice versa.

The experiment results show that Packer has much better space utilization than GC-MC and GC-MS. Also, the parallel compaction algorithms in Packer demonstrate great scalability. Although GC-MS has lower GC pause time than Packer in the sequential case, as the number of threads increases, Packer gradually takes the performance advantage over GC-MS due to better scalability. In addition, we evaluate Packer's impact on the overall performance. Note that although GC only takes about 10% of the total execution time in the application programs, Packer is able to achieve 1.2% and 3% performance gain over GC-MS in the Specjbb and Dacapo benchmark suites, respectively. Hence, our results demonstrate that Packer is highly space-and-time efficient.

Our ongoing work is to apply Packer in more GC designs. Specifically, we intend to implement a generational Packer, which consists of a physical Nursery Object Space (NOS), a virtual Large Object Space (LOS), and a virtual Mature Object Space (MOS). In minor collection, live normal objects are copied from NOS to virtual MOS, and virtual LOS can be marked and swept. Then in major collection, the full heap is compacted. In the next step, we would attempt to manage virtual NOS, virtual MOS, and virtual LOS in one physical space, thereby achieving a generational GC with fully virtualized space management.

REFERENCES

1. P.J. Caudill, A. Wirfs-Brock. A Third Generation Smalltalk-80 Implementation. Conference proceedings on Object-oriented programming systems, languages and applications, Portland, Oregon, USA, 1986
2. M. Hicks, L. Hornof, J.T. Moore, S.M. Nettles. A Study of Large Object Spaces. In Proceedings of ISMM 1998
3. S. Soman, C. Krintz, D.F. Bacon. Dynamic selection of application-specific garbage collectors. In Proceedings of ISMM 2004.
4. D. Barrett and B.G. Zorn. Garbage Collection using a Dynamic Threatening Boundary. In Proceedings of PLDI 1995.
5. R.E. Jones. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.
6. C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In the USENIX JVM Symposium, 2001
7. D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. In the ACM Conference on Object-Oriented Systems, Languages and Applications, 2004.
8. H. Kermany and E. Petrank. The Compressor: Concurrent, incremental and parallel compaction. In *PLDI*, 2006.
9. M. Wegiel, C. Krintz, The Mapping Collector: Virtual Memory Support for Generational, Parallel, and Concurrent Compaction, In ASPLOS '08, Seattle, WA, March 2008.
10. Apache Harmony: Open-Source Java SE. http://harmony.apache.org/
11. Spec: The Standard Performance Evaluation Corporation. http://www.spec.org/.
12. Dacapo Project: The DaCapo Benchmark Suite. http://www-ali.cs.umass.edu/dacapo/index.html
13. Ming Wu and Xiao-Feng Li, Task-pushing: a Scalable Parallel GC Marking Algorithm without Synchronization Operations. IEEE IPDPS2007.
14. J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine. Commun. ACM, 1960 3(4):184~195
15. E. Toshio, K. Taura, A. Yonezawa. A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines. Proc. Of ACM/IEEE conference on Supercomputing, New York, USA, 1997.
16. T. Domani, E.K. Kolodner, E. Lewis, etc. Implementing an On-the-fly Garbage Collector for Java. ACM SIGPLAN Notices, 2001, 36(1):155~166
17. The Mono Project. www.mono-project.com