This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Packer: Parallel Garbage Collection Based on Virtual Spaces

Shaoshan Liu, Jie Tang, Ligang Wang, Xiao-Feng Li, and Jean-Luc Gaudiot, *Fellow, IEEE*

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 normal object space (non-LOS). When either space is full, garbage collection is triggered even though the other space may still have plenty of 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 inefficient on multiple spaces. To address these problems, we propose Packer, a 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 physical space. With multiple virtual spaces, Packer offers efficient memory management. With one physical space, Packer avoids the problem of inefficient space utilization. To reduce the garbage collection pause time, we also propose a novel parallelization method that is applicable to multiple virtual spaces. Specifically, we reduce the compacting GC parallelization problem into a DAG (discreted acyclic graph) traversal parallelization problem, and apply it to both normal and large object compaction.

Index Terms—garbage collection, Java Virtual Machine, memory management, parallel systems

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 garbage collector (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 run-time variation of the 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 better 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. The parallelism seems inadequate when there are few large objects.

In this paper, we propose Packer, a parallel garbage collection algorithm based on the novel concept of virtual spaces. Unlike some conventional garbage collectors [2] which 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. 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 DAG structure. When one of the virtual spaces needs more space, then it searches the DAG 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

- *Shaoshan Liu is with Microsoft. E-mail: shaoliu@microsoft.com*
- *Jie Tang is with the School of Computer Science and Technology, Beijing Institute of Technology. E-mail: tangjie.bit@gmail.com*
- *Ligang Wang is with the Intel China Research Center. E-mail: ligang.wang@intel.com*
- *Xiao-Feng Li is with the Intel China Research Center. E-mail: xiao.feng.li@intel.com*
- *Jean-Luc Gaudiot is with the Department of EECS, University of California, Irvine. E-mail: gaudiot@uci.edu*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

Digital Object Indentifier 10.1109/TC.2011.193

0018-9340/11/$26.00 © 2011 IEEE

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

DAG (discrete acyclic graph) 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 [10], 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. 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

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.

It has been empirically observed that in many programs, the most recently created objects are also those most likely to become unreachable quickly. Generational GCs leverage this property and divides objects into generations [18]. In this case, separate memory regions are used for objects of different generations. For example, the heap can be divided into a nursery object space (NOS) to store newly created objects and a mature object space (MOS) to store mature objects, *i.e.* the objects that survive one or more collections. When NOS becomes full, GC happens in NOS and moves those few live objects to MOS, and the entire NOS region can then be overwritten with fresh objects; and we call this a minor collection. Semi-Space GC is another technique that exploits the temporal localities of objects [29]. In Semi-Space GC designs, the heap memory is divided into two equally-sized regions: the from-space and the to-space. During normal execution, the mutator allocates new objects from from-space. Eventually, continued allocation exhausts from-space causing the program to be suspended while the collector reclaims memory. Using the Semi-Space concept, Fenichel and Yochelson designed a Lisp Garbage Collector [19], in which the heap is divided into two regions. Only one of the two regions is used at any time. Objects are allocated in one region until the space has been exhausted. Then all objects are moved to the other region, being placed side by side, so that there is no memory fragmentation in the newly copied region. Although this scheme is better in pause time, it is worse in space cost than other algorithms. Other well-known GC designs that use this concept include Baker's collector [20], Brooks's collector [21], and the Train collector [22]. Specifically, Baker's and Brooks's techniques use a read barrier, which is not very efficient. The Train collector can run with very low space overheads. It can suffer from large remembered sets, though there are proposals on limiting that space overhead.

Early efforts on designing parallel collectors include Halstead's collector for Multilisp [16]. The Multilisp collector, however, has problem scaling because it does not load-balance the collection work. Cheng and Blelloch [15] introduced a parallel, real-time garbage collector which is designed for shared-memory multiprocessors. It can reduce excessive interleaving, handling stacks and global variables, reduce double allocation and special treatment for large and small objects. By making all aspects of the collector incremental and allowing an arbitrary number of application and collector threads to run in parallel, they were able to achieve tight theoretical bounds on the pause time for any application threads as well as bound the total memory usage. Appel *et al.* presented a parallel copying collector intended to run on conventional machines [24]. Their scheme takes advantage of virtual memory hardware. They require intervention when the page on which an object resides is first *accessed* (either written or read), whereas our scheme requires intervention only when the page is first written, and then only if the operating system does not allow use of hardware dirty bits. Since their algorithm also copies list structures breadth-first, and thus does not preserve locality in list structures, this may result in a flurry of such intervention at the beginning of a collection. Similarly, Demers *et al.* introduced a parallel collection algorithm based on virtual checkpoints implemented with a copy-on-write strategy [25]. The algorithm does not incur the copying overhead, is typically easier to implement, and requires no additional memory. Also, DeTreville introduced a parallel trace-and-sweep collector which uses virtual memory hardware instead of explicit mutator cooperation [26]. His collector requires that slightly less work be performed while the mutator is stopped but, it requires that the collector be notified on initial read accesses by the mutator. In [23], Boehm *et al.* rely on virtual memory hardware to provide information about pages that have been updated or "dirtied" during a given period of time. This method has been used to construct a mostly parallel trace-and-sweep collector that exhibits very short pause times.

Moreover, Endo *et al.* [14] introduced a parallel stop-the-world GC algorithm using work stealing. Their algorithm depends on threads with work copying some work

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

to auxiliary queues, where the work is available for stealing. Threads without work look for an auxiliary queue with work, lock the queue, and steal half of the queue's elements. Halstead describes a multiprocessor GC for Multi-Lisp [27]. Each processor has its own local heap, and they use lock bits for moving and updating forwarding pointers. Load balancing is done statically rather than dynamically. Steensgaard explored a clever method for partially parallelizing collection [28]: Compile-time analysis identifies allocation sites that allocate objects that never *escape* the allocating thread (are never accessible to other threads.) Such objects are allocated in a *thread-local heap,* which can be collected independently of other threads. This technique avoids the synchronization issues that general parallel collection must address, but requires extensive and expensive static analysis, and only a subset of objects may be collected thread-locally. Flood and Detlefs used a lower-overhead work-stealing mechanism [6], and by addressing the harder problem of parallelizing relocating collectors, not just a non-relocating mark-sweep algorithm. They balance the work of root scanning, using *static overpartitioning,* and also to balance the work of tracing the object graph, using a form of dynamic load balancing called *work stealing.* They use this infrastructure to parallelize two well-known collection schemes: a two-space copying algorithm (*semispaces*) and a mark-sweep algorithm with sliding compaction (*markcompact*).

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. This leads to frequent collections and lower performance.

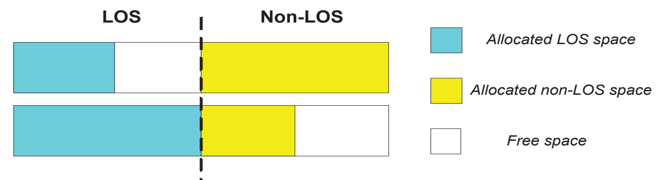

Figure 1: compacting GC (GC-MC) with separate allocation spaces

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 physically 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.

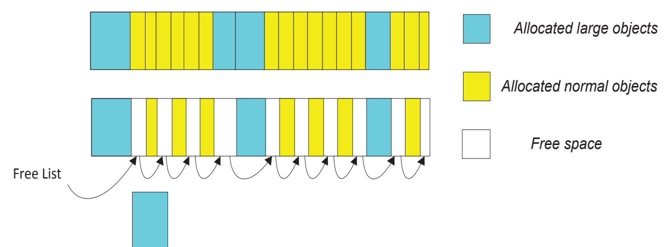

Figure 2: mark-sweep GC (GC-MS)

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

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 until it finds a suitable free region.

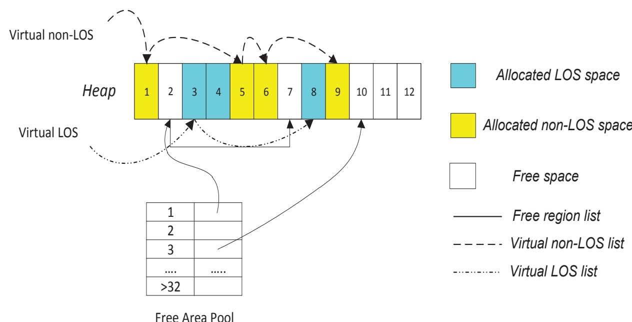

Figure 3: the design of heap structure in Packer

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, blocks 2 and 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, blocks 10, 11, and 12 form a contiguous free region, 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. With this design, the virtual spaces can grow based on need and garbage collection only happens when the whole heap is fully utilized.

3.2 Object Allocation in Packer

When multiple threads are running in an application, 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 tremendous. In most applications, the majority of objects are normal objects which are much smaller than the block size (set to 32 KB by default). 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, 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. As shown in Figure 4: first, it finds out the pointers to the unoccupied region and the boundary of the block. Next it checks whether the block contains enough space to hold the new object. If so, it updates the *free* pointer to point to the new unoccupied region on the block. Otherwise, it returns *NULL* and forces the mutator to fetch another block from the Free Area Pool.

```
1.  free = allocator->free;
2.  ceiling = allocator->ceiling;
3.  new_free = free + size;
4.  if (new_free <= ceiling){
5.    allocator->free= new_free;
6.    return free;
7.  }
8.  return NULL;
```

Figure 4: normal object allocation from the thread local block

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. If both slot 1 and slot >32 are null, then scans down the table and tries to allocate from slot 2, slot 3, and so on. In these cases, it only requires one atomic operation for each thread local block. 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 removing 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. On the other hand, 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.

Different from normal objects, each large object occupies one or more blocks. Thus, the Mutators directly allocate large objects in the Free Area Pool. As shown in Figure 5, when there is an allocation request, a Mutator first checks the number of blocks requested, *block_count*. Then it searches the Free Area Pool to check whether there is any freed region with *block_count* or more free blocks using *block_count* as index. If such a free region can be found, the mutator updates the block status to *BLOCK_USED*, as well as updates the block header information. Otherwise, garbage collection should be triggered.

Figure 6 shows the search algorithm in the Free Area Pool: it requests one free block but slot 1 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

block 1 and stores block 2 back into the Free Area Pool. With this design, Packer can achieve fast object allocation for both the large and normal objects.

```
1.  block_count = NUM_BLOCK_FOR_SIZE(size);
2.   block= free_regions_alloc_block(block_count);
3.   if(block != NULL){
4.     block->status = BLOCK_USED;
5.     block->num_multi_block = block_count;
6.     return block->base;
7.   }
8.   return NULL;
```

Figure 5: large object allocation

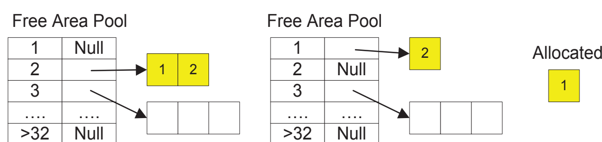

Figure 6: block allocation from the Free Area Pool

3.3 Garbage Collection in Packer

When the heap is fully occupied, 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 Figure 7. 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.

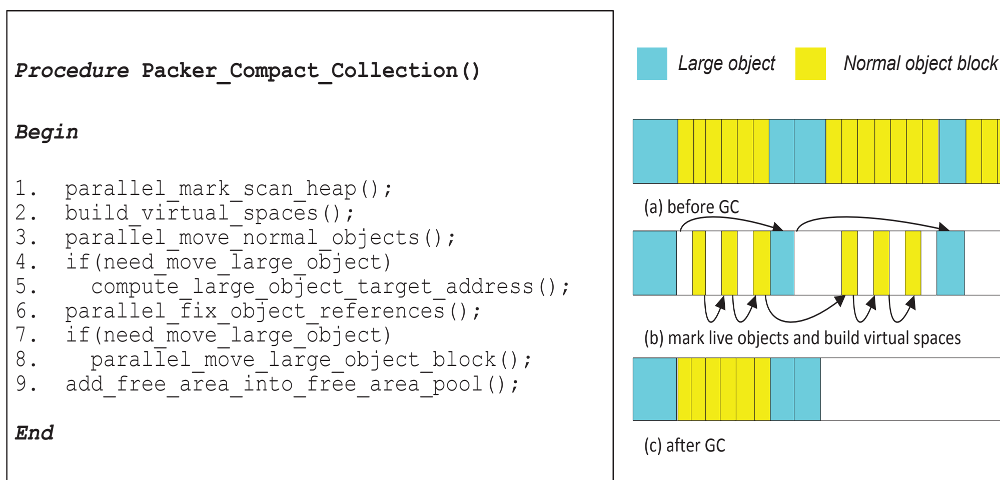

Figure 7: garbage collection algorithm in Packer

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.

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. Pinned objects are the objects that can not be moved or garbage collected. One example of Pinned objects are the communication ports between the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

managed and unmanaged environments. 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. Pinned objects introduce serious problems for GC designs. Imagine that one is using a compaction algorithm to move all live objects towards one end of the heap. After compaction is done, one assumes that the rest of the heap is empty and can be used for object allocation. However, since pinned objects can not be moved, they may reside in the free regions of the heap, and later it may be overwritten by the newly allocated data.

In conventional compacting GC designs, additional management techniques are utilized to deal with pinned objects in the heap. Therefore, 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. We will explore these extensions of Packer in our future work.

4 Parallelization of Garbage Collection

In this section we first demonstrate how we reduce the compaction parallelization problem into a DAG traversal parallelization problem. Then we present the implementation of parallel normal and large object compaction in Packer, as well as the load balance mechanisms.

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.

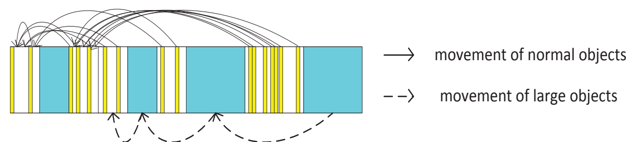

Figure 8: normal and large object compaction

As shown in Figure 8, 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. Note that in this context, dependence means location dependence, such that object B needs t be moved to the location where object A occupies, therefore, object B cannot be moved to object A's location unless object A has been moved elsewhere. In this situation, we say that object B has a dependence on object A.

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. When there are only few large objects, the parallelism is seemingly inadequate.

This observation indicates that we need to set a proper parallelization granularity to reduce 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 such 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, greatly reducing the synchronization overhead. 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 9, 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 blocks 7 and 5, so the dependencies between blocks 8 and 5 are false data dependencies. Block 8 can be moved independently of block 7.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

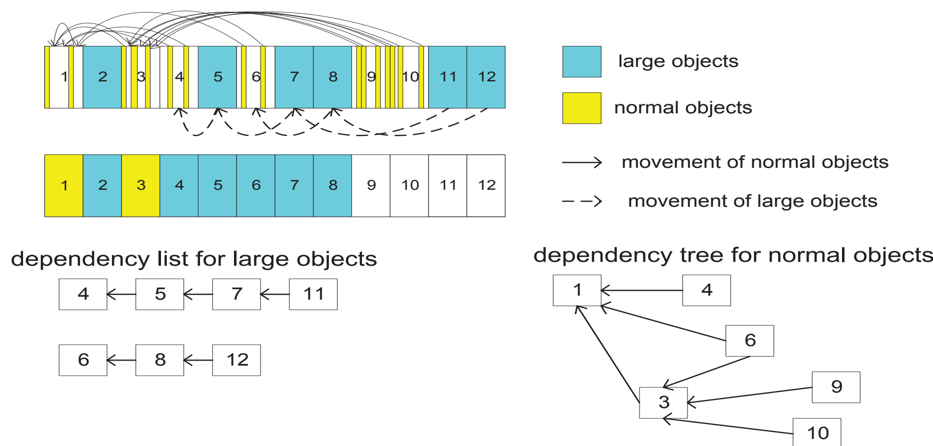

Figure 9: block-based heap structure

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 DAG, such as the one in Figure 9, 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 DAG to obtain a source block and a target block. After the current data movement is done, the thread moves down the DAG 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 DAG. We have reduced the compaction parallelization problem into a DAG 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 DAG degenerates into dependence lists.

4.2 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. Before collection starts, a number of disjoint dependence lists are generated to capture the dependence relationship among the large object blocks. The pseudo-code of the dependency lists generation algorithm is shown in Figure 10: first, multiple collectors compete to grab the large objects from the heap; the accesses to the heap are guarded with atomic operations (Label 1). Second, after obtaining a task, the collector thread updates the *global_target_address* to allow other threads to continue (Label 2). At last, the collector computes the dependencies between the source and target blocks and inserts these blocks into the dependency lists (Label 3).

```
global_target_address = heap_start;
for (each collector thread in parallel ){
 Label1: // grab a large object
   large_obj = pick_node_atomically(large_object_list);
   obj_size = num_of_blocks (large_obj) * size_of_block;
 Label2: //increment global_target_adress
   do{
     old_target_address = global_target_address;
     new_target_address = old_target_address + obj_size;
     temp  = atomic_compxchg (global_target_address,
                new_target_address, old_target_address);
   }while( temp !=  old_target_address);
 Label3: // build the dependency list
   source_block = address_to_block_index( large_obj);
   target_block = address_to_block_index( old_target_address);
   for( i = 0; i++; i < num_of_blocks (large_obj) ){
     insert_a_dependence_to_list(target_block, source_block);
     target_block++;
     source_block++;
   }
} //loop back for next object
```

Figure 10: dependence list generation

Figure 11 shows the pseudo-code of the parallel compaction process: each collector atomically grabs 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

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.

```
Procedure Parallel_Large_Object_Compaction()
Begin
1.      dep_list = get_next_compact_dep_list();
2.      while(dep_list){
3.       target_block = get_first_block(dep_list);
4.       source_block = get_next_block(dep_list);
5.       while(source_block != NULL){
6.         memmove(target_block, source_block);
7.         target_block = source_block;
8.         source_block = get_next_block(dep_list);
9.        }
10.      dep_list = get_next_compact_dep_list();
        }
End
```

Figure 11: parallel large object compaction

4.3 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.

Phase 1: Live object marking. It traces the heap from root set and marks all the live objects;

Phase 2: Object moving. It copies the live objects to their new locations;

Phase 3: 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. The sector size is the same size as the page size, which is usually 4 KB. 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 DAG 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 DAG. 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 DAG 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 DAG is generated dynamically instead of pre-generated.

4.4 Load Balance

The parallel compaction algorithms would achieve high performance only if the workload for each thread is balanced. However, load imbalance can occur in dependency lists and DAG. As shown in Figure 12, if there are two threads working on compaction, the work load of thread 1 would be much higher than that of thread 2 due to the existence of a long dependency list. In this case, thread 2 has to wait until thread 1 finishes its task.

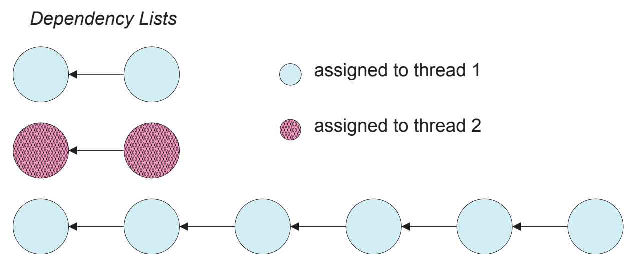

Figure 12: load imbalance in depdency lists

To enhance parallelism, we introduce the load balance algorithms in this section. First, we have implemented a heuristics that counts the total number of dependency lists and divides them into *N* (number of threads) chunks and then collapses each chunk into a dependency list. We call this approach *Task Collapse*, the main advantage of this approach is its simplicity. It simply counts the number of dependency lists and divide the dependency lists equally among all collector threads. Thus, it does not have to traverse all dependence lists to count the number of blocks. However, the disadvantage of this design is that if the dependency lists are highly imbalanced, *Task Collapse* may not perform well because it may collapse several long lists into one list and several short lists into another.

In cases where the dependency lists are highly imbalanced. We can use a more sophisticated *Task Pushing* load balance approach [13]. For instance, when a dependency list gets too long or when there are always one or two disjoint sub-DAGs generated after the root block is filled, the work load between threads would be imbalanced. Our *Task Pushing* design can solve this problem by using virtual target blocks. To break long lists, we can use a virtual target block for one of the blocks in the middle of the list, and thus this virtual target block breaks the long list into two halves. The virtual target block also serves as the new root of the newly created dependency list. To implement this virtual target block, we move the source block to a reserved region (the virtual block) such that the source block no longer depends on another block and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

becomes a root block. Then after compaction finishes, this block in the reserved region (the virtual block) can be thrashed. For more details of the design and implementation of *Task Pushing*, please refer to [13].

For instance, in Figure 12, the third dependency list is too long. Using *Task Pushing,* we can break the third dependency list into two halves, such that we move the third block to a reserved region *rsv* and make *rsv* depend on the second block, so *rsv* becomes the tail of third dependency list. Next, since now we have two copies of the third block, the third block no long depends on the second block, thus we create a new dependency chain to hold the third block and whatever comes after it.

This algorithm can be applied to dependency DAG as well, but instead of breaking a long dependency list, we break a dependency DAG into sub-DAGs and assign each sub-DAG to a thread. The pseudo-code is shown in Figure13, where *Wi* is the local working set of collector *Ci*. This algorithm applies the idea of *Task Pushing*, such that a collector pushes its excessive tasks to other idle collectors (line 17). When all the collectors have no more tasks, the execution finishes. Otherwise, the collectors will loop back to check if other collectors have pushed new tasks to their local working sets.

```
1.  while(working set Wi is not empty){
2.    Node_root = get_node_from_set(W_i);
3.    foreach (Node_child  Node_root's children) {
4.      move_data(Node_root, Node_child);
5.      decrement num_of_parents of Node_child;
6.      if (num_of_parents of Node_child == 0)
7.        put_node_to_set(W_i, Node_root)
8.      }
9.      remove Node_root from the tree;
10.     if(W_i is empty) break;
11.     foreach (collector C_k  other collectors){
12.       if (collector C_k has no task){
13.         if (W_i has only one tree){
14.           break it into subtrees;
15.         }
16.         Node_root = get_node_from_set(W_i);
17.         put_node_to_working_set(W_k, Node_root);
18.       }
19.     }
20.   }
21.   if (all collectors come to here) // barrier
22.     exit;
23.   else goto step 1
```

Figure 13: load balance algorithm for parallel compaction

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 14 shows the space utilization of different designs. Note that in this paper we define space utilization as the percentage of heap space usage when GC occurs. For example, if the heap size is 512 MB, and at the time when GC occurs, 500 MB of the heap space is used and 12 MB of the heap space is free, then the space utilization is 97.7%. The results show that 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, 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,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

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.

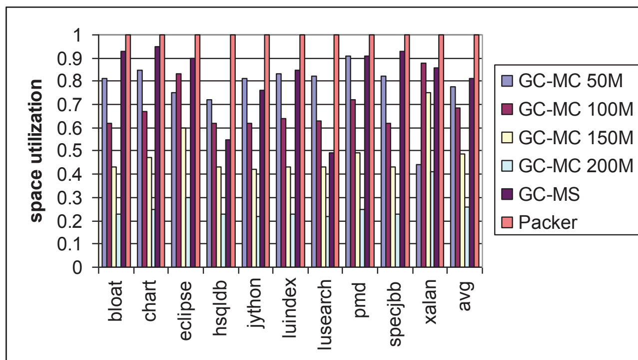

Figure 14: space utilization of GC-MC, GC-MS, and Packer

5.2 Sequential Performance

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 *pmd*, in which GC-MS's pause time is only 1/ 11 of that of Packer. This is because Mark-Sweep does not involve any object movement, which may incur a high performance overhead.

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	
bloat	*126*	*1236*	*133*	*206*	*189*	*1557*	*236*	*1573*	*355*	*1780*	*756*	*2568*
chart	*51*	*601*	*52*	*69*	*59*	*605*	*78*	*642*	*117*	*700*	*234*	*883*
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*
luindex	*11*	*87*	*13*	*10*	*14*	*90*	*18*	*97*	*27*	*105*	*53*	*127*
lusearch	*84*	*3389*	*175*	*3465*	*100*	*3707*	*134*	*4323*	*203*	*5553*	*416*	*9331*
pmd	*74*	*982*	*74*	*87*	*83*	*1002*	*100*	*1036*	*148*	*1135*	*290*	*1392*
SPECjbb	*39*	*1204*	*41*	*1160*	*119*	*3927*	*F*	*F*	*F*	*F*	*F*	*F*
xalan	*324*	*1578*	*285*	*616*	*700*	*233*	*283*	*1506*	*329*	*1605*	*707*	*2349*

5.3 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. As shown in Figures 15 and 16, the Y-axis of these figures represents the normalized total GC pause time. Figure 15 shows Packer's scalability. In general, Packer demonstrates very good scalability. On average, the speedups of Packer are 1.92x, 2.64x, and 3.15x respectively with 2, 3, 4 collectors.

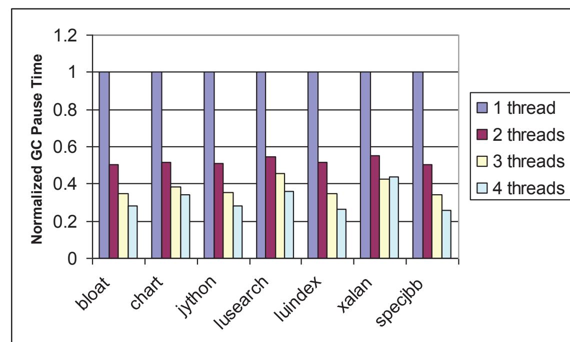

Figure 15: Packer scalability

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

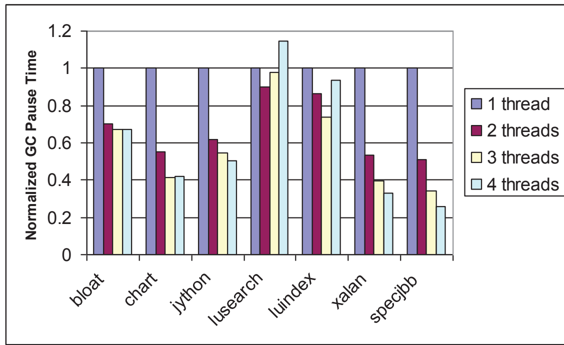

Figure 16: mark-sweep scalability

Figure 16 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.

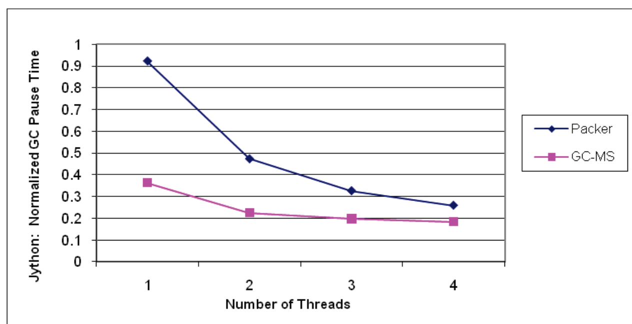

Figure 17: comparison of parallel Packer and GC-MS

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 17 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.

5.4 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 18 shows the results on SPECjbb. 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. To generate these results, we repeated the experiments for ten times and presented the average results. Packer's performance is consistently 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 would translate into 12% GC performance improvement, which is a significant amount. 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.

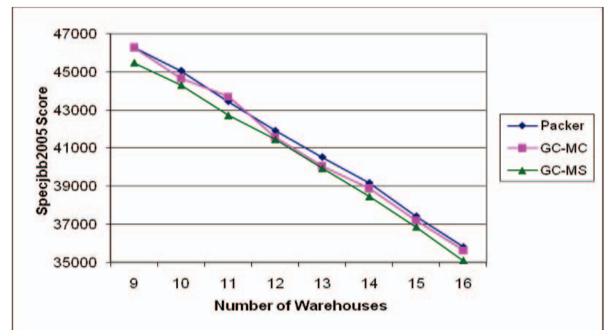

Figure 18: impacts on SPECjbb 2005 overall performance

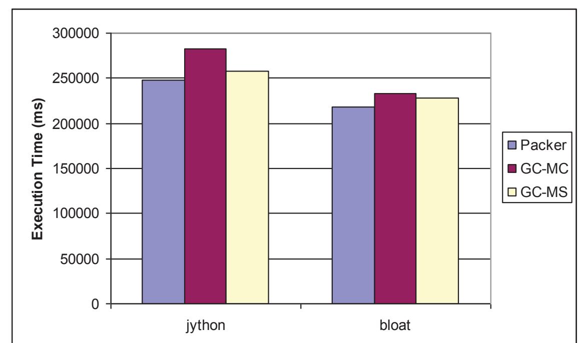

Figure 19: impacts on Dacapo overall performance

Figure 19 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,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

compaction can be more efficient.

5.5 Load Balance

In our study on LOS load balance with xalan, we found that the max length of a dependency list was 48, while the majority (78%) of dependency lists contained only one moving task (only one source block and one target block). This result has two implications: First, without optimization, the dependency lists may be imbalanced such that there were several long lists and a large amount of short lists, and the long lists became the performance bottleneck since they could only be executed sequentially. Second, it required an atomic operation to fetch a dependency list, when the list contained only one block, then the performance gain could be very low. Actually, we found out that this overhead was 38%, that is, if the task takes 100 cycles to move a block, then the synchronization overhead to fetch this task is 38 cycles on average.

In another study on non-LOS load balance with SPECjbb2005 benchmark, we found out that the maximum depth of a dependence DAG was 353 while the average depth was only 22.3. Also, the maximum number of child nodes for each node was 20 while the average was 1.5. This implies the possibility of the existence of some huge DAG that contained a large number of nodes, along with some small DAGs that contained only few nodes.

To address these problems, in Section 4.4 we have introduced two load balance algorithms *Task Collapse* and *Task Pushing*. In our experiments, we found that *Task Collapse* is sufficient for most benchmark programs. Since *Task Pushing* needs to go through all dependency lists, it introduces a fairly high overhead. It would bring performance gain only when its overhead can be significantly amortized. Therefore, *Task Pushing* is suitable for large server applications that require a large heap size (~10 GB).

6 Conclusions and Future Work

Space and time efficiency are the two most important design goals in GC 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 DAG traversal problem, and then designed solutions to eliminate false sharing and to reduce the synchronization overhead. 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, which translates into about 12% and 30% reduction of GC times, respectively. Hence, our results demonstrate that Packer is highly space-and-time efficient.

Our ongoing work is three-fold: first, we plan 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.

Second, we plan to test the proposed parallel compaction algorithms on high-end commercial servers that consist of tens of cores and >10 GB of memory. A key challenge in this new setting is to maintain the scalability of these algorithms, thus one essential component is the load balance algorithm, such as *Task Pushing*.

Finally, as discussed in subsection 3.4, the Packer concept can be applied to pinned object management, the management of managed and native data in the same heap, and the management of discrete physical areas. We aim to apply the virtual spaces design in these areas.

Acknowledgements

This work is partly supported by the National Science Foundation under Grant No. CCF-1065448. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

[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] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines *Proceedings of High Performance Networking and Computing (SC97)*, 1997.

[15] P. Cheng, G.E. Blelloch, A Parallel, Real Time Garbage Collector, ACM SIGPLAN Notices, Volume 36, 2001.

[16] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems, 1985.

[17] The Mono Project. www.mono-project.com

[18] W. Appel, Simple generational garbage collection and fast allocation. *Software. Practice & Experience*. 19, 1989

[19] R.R. Fenichel, J.C. Yochelson, A Lisp Garbage- Collector for Virtual-Memory Computer Systems, Communications of the ACM, 1969, 12(11), pp. 611–612.

[20] H.G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[21] R.A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Proceedings of *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, TX, August 1984.ACM Press.

[22] R.L. Hudson and J.E.B. Moss. Incremental garbage collection for mature objects. In Proceedings *of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[23] H.J Boehm, A.J. Demers, S. Shenker, Mostly Parallel Garbage Collection, ACM SIGPLAN Notices, 1991.

[24] A. Appel, J.R. Ellis, and K. Li, "Real-time Concurrent Collection on Stock Multiprocessors", *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, SIGPLAN Notices 23*, 7 (July 88), pp. 11-20.

[25] Demers. A., M. Weiser, B. Hayes, H. Boehm, D. Bobrow, S. Shenker, "Combining Generational and Conservative Garbage Collection: Framework and Implementations", *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, January 1990, pp. 261-269.

[26] J. DeTreville, "Experience with Concurrent Garbage Collectors for Modula-2+", Digital Equipment Corporation, Systems Research Center, Report No. 64.

[27] R.H. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *1984 ACM Symposium on LISP and Functional Programming*, 1984. ACM.

[28] B. Steensgaard. Thread-specific heaps for multi-threaded programs. *ACM SIGPLAN Notices*, January 2001

[29] C. J. Cheney. A non-recursive list compacting algorithm. Communications of the ACM,13(11):677 8, November 1970

Shaoshan Liu is currently with Microsoft. He received Ph.D. in Computer Engineering, M.S. in Biomedial Engineering, M.S. in Computer Engineering, and B.S. in Computer Engineering, respectively in 2010, 2007, 2006, and 2005 respectively, all from the University of California, Irvine. His research interests include parallel computer architectures, embedded systems, runtime systems, as well as biomedical engineering.

Jie Tang is a Ph.D. candidate in Beijing Institute of Technology, China. Her research interests include high performance computer architecture, cloud computing, and embedded system. Jie's Ph.D. thesis title is "Performance Acceleration and Energy Efficiency Mechanisms in Cloud Computing Environment", in which she studies the impact of hardware acceleration and prefetching techniques in enhancing both the performance and energy efficiency for cloud computing environment. During her Ph.D. study, Jie also worked as a visiting researcher in the Center for Embedded Computer Systems, University of California, Irvine. Jie holds a B.S. in Computer Science from National University of Defense Technology, China.

Ligang Wang is currently an R&D engineer at the Managed Runtime Technologies Center of Intel China Research Center. His research interests focus on Garbage Collection (GC), Java Virtual Machine (JVM), and other runtime technologies. Ligang has developed important algorithms for the garbage collector module of Apache Harmony. Ligang graduated from University of Science and Technology of China with Ph.D degree at 2006. His dissertation was on real time scheduling and operating system design.

Xiao-Feng Li's research interests are with programming systems and language design. Currently Xiao-Feng is the manager of China Runtime Technologies Lab in Intel China Research Center. His team works on runtimes including Java Virtual Machine, scripting engines, browser technologies and their impacts on computer architecture. Before joining Intel, Xiao-Feng worked with Nokia Research Center. Xiao-Feng has a Ph.D degree in computer science.

Jean-Luc Gaudiot received the Diplôme d'Ingénieur from the École Supérieure d'Ingénieurs en Electrotechnique et Electronique, Paris, France in 1976 and the MS and PhD degrees in Computer Science from the University of California, Los Angeles in 1977 and 1982, respectively. He is currently a Professor of the Electrical and Computer Engineering Department at the University of California, Irvine. Prior to joining UCI in January 2002, he was a Professor of Electrical Engineering at the University of Southern California since 1982, where he served as and Director of the Computer Engineering Division for three years. He has also done microprocessor systems design at Teledyne Controls, Santa Monica, California (1979–1980) and research in innovative architectures at the TRW Technology Research Center, El Segundo, California (1980–1982). He consults for a number of companies involved in the design of high-performance computer architectures. His research interests include multithreaded architectures, fault-tolerant multiprocessors, and implementation of reconfigurable architectures. He has published over 170 journal and conference papers. His research has been sponsored by NSF, DoE, and DARPA, as well as a number of industrial

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

organizations. In January 2006, he became the first Editor-in-Chief of IEEE Computer Architecture Letters, a new publication of the IEEE Computer Society, which he helped found to the end of facilitating short, fast turnaround of fundamental ideas in the Computer Architecture domain. From 1999 to 2002, he was the Editor-in-Chief of the IEEE Transactions on Computers. In June 2001, he was elected chair of the IEEE Technical Committee on Computer Architecture, and re-elected in June 2003 for a second two-year term.He is a member of the ACM, of the ACM SIGARCH, and of the IEEE. He has also chaired the IFIP Working Group 10.3 (Concurrent Systems). He is one of three founders of PACT, the ACM/IEEE/IFIP Conference on Parallel Architectures and Compilation Techniques, and served as its first Program Chair in 1993, and again in 1995. He has also served as Program Chair of the 1993 Symposium on Parallel and Distributed Processing, HPCA-5 (1999 High Performance Computer Architecture), the 16th Symposium on Computer Architecture and High Performance Computing (Foz do Iguaçu, Brazil), the 2004 ACM International Conference on Computing Frontiers, and the 2005 International Parallel and Distributed Processing Symposium.In 1999, he became a Fellow of the IEEE. He was elevated to the rank of AAAS Fellow in 2007.