Int J Parallel Prog (2011) 39:451–472
DOI 10.1007/s10766-010-0151-4

Space-and-Time Efficient Parallel Garbage Collector for Data-Intensive Applications

Shaoshan Liu · Ligang Wang · Xiao-Feng Li · Jean-Luc Gaudiot

Received: 7 June 2010 / Accepted: 15 September 2010 / Published online: 1 October 2010

Abstract As multithreaded server applications and runtime systems prevail, garbage collection is becoming an essential feature to support high performance systems, especially those running data-intensive applications. The fundamental issue of garbage collector (GC) design is to maximize the recycled space with minimal time overhead. This paper proposes two innovative solutions: one to improve space efficiency, and the other to improve time efficiency. To achieve space efficiency, we propose the Space Tuner that utilizes the novel concept of allocation speed to reduce wasted space. Conventional static space partitioning techniques often lead to inefficient space utilization. The Space Tuner adjusts the heap partitioning dynamically such that when a collection is triggered, all space partitions are fully filled. To achieve time efficiency, we propose a novel parallelization method that reduces the compacting GC parallelization problem into a tree traversal parallelization problem. This method can be applied for both normal and large object compaction. 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. Our proposed algorithm overcomes the difficulties by dividing the heap into equal-sized blocks and parallelizing the movement of the independent blocks. It

S. Liu (✉) · J.-L. Gaudiot
University of California, Irvine, CA, USA
e-mail: shaoshal@uci.edu

J.-L. Gaudiot
e-mail: gaudiot@uci.edu

L. Wang · X.-F. Li
Intel China Research Center, Beijing, China
e-mail: ligang.wang@intel.com

X.-F. Li
e-mail: xiao.feng.li@intel.com

is noteworthy that these proposed algorithms are generic such that they can be utilized in different GC designs. The proposed techniques have been implemented in Apache Harmony JVM and we evaluated the proposed algorithms with SPECjbb and Dacapo benchmark suites. The experiment results demonstrate that our proposed algorithms greatly improve space utilization and the corresponding parallelization schemes are scalable, which brings time efficiency.

Keywords Java virtual machine · Garbage collection

1 Introduction

Garbage collection technology is widely used in managed runtime systems such as Java virtual machine (JVM) [2] and Common Language Runtime (CLR) [6]. Data-intensive multithreaded applications with large heaps running on modern servers present new challenges as far as designing suitable garbage collectors is concerned. Particularly, server applications are required to operate continuously and remain highly responsive to frequent client requests. Thus the garbage collector should impose minimum pause time while providing maximum throughput. On the other hand, increasingly parallel multicore systems will be used even in low-end devices that impose real-time constraints. When garbage collection is used in these systems, its impact on the overall performance needs to be minimized so as to meet the real-time constraints. Meanwhile, garbage collection needs to be effective in recycling space, especially when the available memory space is limited. Therefore, to meet the needs of modern and future data-intensive multithreaded applications, it is essential to design garbage collectors that provide both space and time efficiency.

There are two main aspects in garbage collector (GC) design, namely, the partitioning of heap space and the algorithms for garbage collection. As shown in [13], for better 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 statically. The current known GCs with separate allocation spaces mostly suffer from the problem that they don't fit well with the dynamic variations of object size distribution at runtime. This problem leads to imbalanced space utilization and impacts the overall GC performance negatively. Besides LOS/non-LOS space partitioning, the problem of imbalanced space utilization indeed exists in any GCs that have multiple spaces. For example, a generational GC has typically a young object space and a mature object space. The space partitioning for both spaces also needs a careful design to achieve maximal GC efficiency.

For garbage collection algorithms, conventional mark-sweep and reference counting collectors are susceptible to fragmentation due to the lack of object movements. To address this problem, copying or compacting GCs are introduced. Compaction algorithms are now widely utilized in GC designs [14,1,15,19]. Compaction eliminates fragmentation 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, none of the proposed parallel compaction algorithms can be used for large object compaction. 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, when there are only few very large objects, the parallelism is seemingly inadequate.

The fundamental issue of garbage collector (GC) design is to maximize the recycled space with minimal time and memory overhead. In this paper, we propose two innovative solutions, one to improve space efficiency, and the other to improve time efficiency. To improve space utilization, we introduce the Space Tuner, which adjusts the heap partitioning between the spaces dynamically according to the application's runtime behavior, such that when collection happens, both spaces are fully utilized. To improve time efficiency, we propose a parallelization algorithm that reduces the compaction parallelization problems into a tree traversal parallelization problem, and apply it to both normal and large object compaction. Note that although we demonstrate our algorithms on a parallel compacting GC based on the LISP2 design [14,2], these algorithms are indeed generic and have much broader applications: The Space Tuner can be utilized in any design with multiple allocation spaces, and the parallel compacting algorithms can be applied in any design that involves object movements.

In this paper we present the design details of the proposed algorithms and evaluate their efficiencies. These algorithms are implemented in Apache Harmony, a production-quality open source JAVA SE implementation [2]. This paper is organized as follows. Section 2 discusses the related work. Section 3 introduces the basic algorithm designs of the Space Tuner and the parallel compaction mechanisms. Section 4 presents the implementation details of these designs in Apache Harmony. Section 5 evaluates and discusses the design with representative benchmarks. And at the end, Sect. 6 concludes and discusses the future work.

2 Related Work

For better memory management, Caudill and Wirfs-Brock first proposed 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 [5]. Hicks et al. have done a thorough study on large object spaces [13]. 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 Soman et al. [16], discussed about applying different GC algorithms in the same heap space, but their work does not involve adjusting the heap partitioning dynamically. The study done by Barrett and Zorn [4] is the only known publication that studies space boundary adjustment. Nevertheless, this work is not aimed for GC efficiency; instead, it is designed to meet the resource constraints such as pause time, thus it attacks completely different problems than our design. To address the inefficiency of space utilization, the Space Tuner utilizes allocation speed to adjust the heap partitioning at runtime.

As exemplified by the LISP2 algorithm [14], compaction algorithms are utilized in GC designs to avoid heap fragmentations. It achieves this by grouping live objects together in the heap and freeing up large contiguous spaces available thereafter for future allocation. However, compaction usually imposes lengthy pause time. To address this issue, several parallel compaction algorithms have been proposed. Flood et al. [12] presented 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. [1] proposed a three-phase parallel compactor that uses a block-offset array and mark-bit table to record the live objects moving distance in blocks. The disadvantage of this algorithm is that it wastes about 3% of heap space due to internal fragmentation of blocks. Kermany and Petrank [15] proposed the Compressor that requires two phases to compact the heap; also Wegiel and Krintz [19] designed the Mapping Collector with nearly one phase. Both approaches strongly depend on the virtual memory support from the underlying operating system. The former leverages the memory protection support to copy and adjust pointer references on a fault, and the latter releases the physical pages that have no live data. Although these algorithms are efficient in compacting normal objects, as far as we know, no algorithm has been proposed to parallelize the compaction of large objects. We hereby propose a novel parallelization method that reduces the normal and large object compaction parallelization problem into a dependence tree traversal parallelization problem.

Concurrent garbage collectors run in parallel to the application on a separate thread using part of the overall computing resources, while the application continues to run on the rest of these resources. Steele and Dijkstra proposed the first concurrent collectors, which are based on mark-sweep algorithms [18,8]. Endo et al. proposed a mostly concurrent collector that does not use compiler supports, such as write barriers, but uses virtual memory primitives [11]. Several groups [9,10,3] have proposed fully concurrent on-the-fly mark-sweep collectors that provide low pause time; but these designs are complex to implement due to the requirement of expensive write barriers. However, 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.

3 Space-and-Time-Efficient Garbage Collection Algorithms

In this section, we conceptually demonstrate the space and time efficiency problems faced in GC designs. To address these problems, we present the Space Tuner algorithm, which aims to improve space efficiency; and the parallel compaction algorithm, which aims to improve time efficiency. The implementation details of the algorithms proposed in this section will be discussed in Sect. 4.

3.1 The Space Tuner

As shown in Fig. 1a, when the heap is partitioned into multiple spaces, for instance LOS and non-LOS, a collection is triggered when either space is full. In times when a collection is triggered by one space while the other space is sparsely filled, the heap is not fully utilized. Consequently, it leads to more frequent collections and lower application performance.

The key question here is why one space would get full before the other one does. This is because one space allocates objects faster than the other. That is, within the same amount of time, a higher fraction of one space's free region is allocated than that of the other space. Hence, if both spaces allocate the same fraction of its free space within a fixed amount of time, then both spaces should be full when garbage collection is triggered. Based on this observation, we propose the Space Tuner. The Space Tuner dynamically monitors the allocation speed, which is defined as the size of objects allocated per unit time (e.g. bytes/seconds), of different spaces, and utilizes this information to adjust the heap partitioning. Thus, in the ideal case, if the sizes of LOS and non-LOS are set proportionally to their respective allocation speeds, then we can guarantee that both spaces become full at the same time. During an application's execution, both spaces may have some live objects surviving after a collection. So the space size that matters is the remaining free (unallocated) size in the space after a collection. The Space Tuner manages to adjust the heap partitioning right after every collection, by allocating the remaining free space of the heap to LOS and non-LOS. The sizes of the free spaces given to LOS and non-LOS are proportional to their allocation speeds.

Figure 1 illustrates the basic idea of the Space Tuner. In the upper half of Fig. 1a, non-LOS becomes full and triggers a garbage collection, but at this time, only 50% of LOS is utilized, thus the allocation speed in non-LOS is twice of that in LOS.

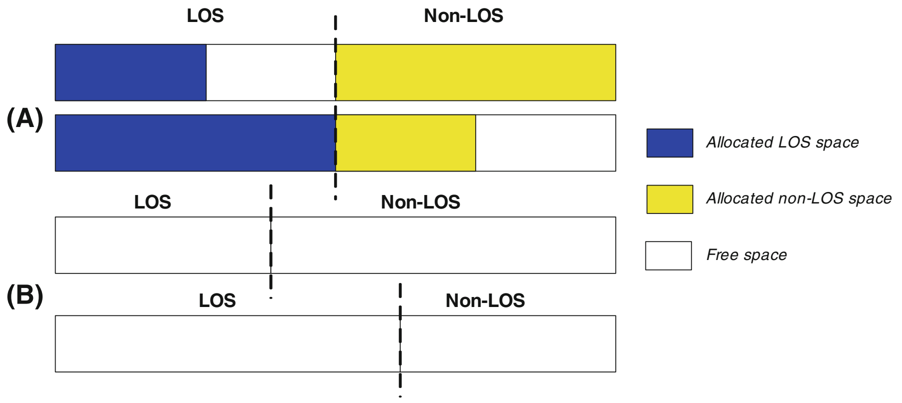

Fig. 1 **a** Space inefficiency in GC with separate allocation spaces. **b** The Space Tuner

In the upper half of Fig. 1b, the Space Tuner captures this information and assigns more space to non-LOS and shrinks LOS correspondingly. Such that in the next allocation period, non-LOS would have more space for object allocation.

$$SpaceSize_{\text{LOS}} = \frac{AllocSpeed_{\text{LOS}}}{(AllocSpeed_{\text{LOS}} + AllocSpeed_{non-LOS})} * (FreeSize_{\text{LOS}} + FreeSize_{non-LOS}) + SurvivorSize_{\text{LOS}} \tag{1}$$

Equation 1 presents the Space Tuner algorithm for LOS size computation. In this equation, $SpaceSize_{\text{LOS}}$ represents the space size of LOS; $AllocSpeed_{\text{LOS}}$ represents the allocation speed of LOS; $AllocSpeed_{non-LOS}$ represents the allocation speed of non-LOS; $FreeSize_{\text{LOS}}$ represents the freed space size in LOS after the collection; $FreeSize_{non-LOS}$ represents the freed space size in non-LOS after the collection; and $SurvivorSize_{\text{LOS}}$ represents the used space size in LOS by the survivors after the collection. This equation calculates the new LOS size for next allocation period based on allocation speed and other information. The first term on the right hand side calculates the relative allocation speed of LOS. The second term calculates the total size of unallocated space in both partitions, such that the unallocated space can be re-assigned to the partitions according to their relative allocation speed calculated in the first term. Then the third term adds the survivor size of the partition to calculate the new size of the partition (LOS in this case). Note that the computation of allocation speed can be flexible. For example, it can be just the total allocated bytes from last collection, or the average value of the speeds in last few collections. In our experience, it is sufficient to use the allocated bytes from the last collection. To get the new non-LOS size, we can simply subtract $SpaceSize_{\text{LOS}}$ from total heap size.

3.2 The Basic Parallel Compaction Algorithm

Compacting GCs move live objects towards one end of the heap to eliminate fragmentations. 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 cost low.

As shown in Fig. 2, there are two problems in the parallelization of object compaction: In non-LOS, there are many normal objects and the data dependencies between

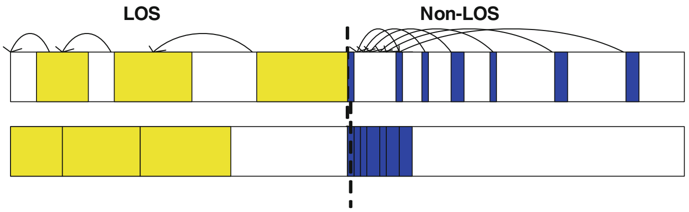

Fig. 2 Normal and large object compaction

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 be well over the performance gain. On the other hand, there exist strong data dependencies in 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 such that it reduces the high synchronization overheads caused by fine-grain data movement (as in non-LOS) and the false data dependencies caused by coarse-grain data movement (as in LOS). Our design is to divide the heap into equal-sized blocks thus the parallelization granularity is a block. For non-LOS, each block contains multiple objects. During collection, each thread obtains a block and moves all the objects in the block. When it finishes the current block, it fetches another one. In this case, at most one atomic operation is required for the movement of multiple objects in a block, thus greatly reducing the synchronization overhead. Furthermore, the atomic operations can be largely eliminated if a group of blocks are assigned to one collector. This is the case in our design shown in later text. On the other hand, for 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 Fig. 3a, originally, due to false data dependencies, blocks 6, 7, and 8 cannot be moved until block 2 has been moved out. With equal-sized blocks, dependencies only exists between blocks 6 and 2, thus the movements of block 7 and 8 can be parallelized. The block sizes for LOS and non-LOS can be different, while they are equal within either space. This brings flexibilities so that we can select appropriate block sizes for either space. For example, the block size of LOS can be as small as possible to bring more parallelism as long as the synchronization overhead can be amortized by the parallelization benefits.

Further complications exist in parallelizing the compaction process. For 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 non-LOS, multiple source blocks may compact into one target block, and thus the access to this target block should be coordinated or synchronized. In order to achieve high performance, the complicated 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 Fig. 3b, which captures all data dependencies between the blocks. For instance, in LOS, block 2 is the source block for block 1 and it is also the target block for block 6. Thus, block 6 cannot be moved to block 2 until block 2 has been moved to block 1. In non-LOS, both block 9 and 10 are the target blocks for block 12, and block 9 is also the target block for block 10. Thus, block 12 cannot be moved to block 10 until block 10 has been moved to

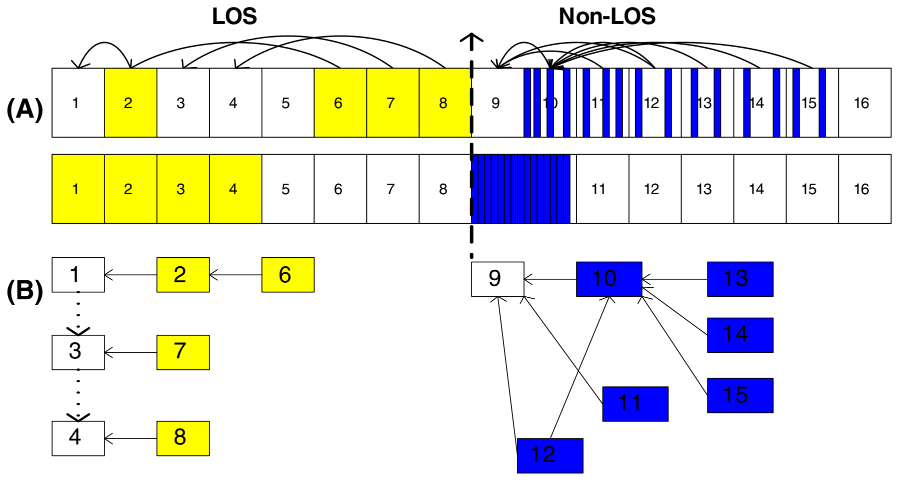

Fig. 3 Partitioning of the heap into equal-sized blocks

block 9. When compaction starts, the threads traverse the tree in parallel, such that each thread obtains 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. Therefore, we actually reduce the compaction parallelization problem into a tree traversal parallelization problem. For LOS compaction, the situation is simpler because one source block has only one target block, and vice versa. Thus in this case the dependency trees degenerate into dependency lists.

4 Implementation Details

All proposed algorithms have been implemented in Apache Harmony, a production-quality open source JAVA Virtual Machine [2]. The default garbage collector in Apache Harmony is a generational garbage collector, such that for normal object allocation, there is a nursery object space (NOS), which is used for new object allocation, and a mature object space (MOS), which is used for the storage of older objects that have survived one or more garbage collections. For better memory management, Apache Harmony GC utilizes LOS for large object management, and non-LOS for normal object management, in this case, non-LOS is MOS and NOS. During allocation, when NOS is full, live objects in NOS are copied to MOS, and LOS is not mark-swept. This is called minor collection. When either MOS or LOS is full, a major collection is triggered to compact in both MOS and LOS. To exploit more parallelism, we divided the heap space into equal-sized blocks, and each block contains a block header for its metadata, including block base address, block ceiling address, block state, etc. In this study, the block size is set to 4 KB and the size threshold for large objects is set to 2 KB.

4.1 The Space Tuner

We implemented the space tuner in Apache Harmony GC such that when major collections happen, it can adjust the LOS and non-LOS sizes based on their respective allocation speeds. The detailed algorithm is illustrated in Fig. 4. When GC receives an allocation request, it first checks whether it is a large object. If it is a large object, its size is added to LOS allocation speed; otherwise, its size is added to non-LOS allocation speed. Since the time elapsed between two collections is the same for both LOS and non-LOS, and the Space Tuner cares only the ratio of their allocation speeds, thus the total allocation size (in unit of *bytes*) in respective spaces can be utilized as their allocation speed. When garbage collection is triggered, the Space Tuner calculates the total survivor size, or the total size of all live objects, in both spaces. Next, the free size in each space is calculated by subtracting the survivor size from the space size, and these numbers are then plugged into Eq. 1 to calculate the new space sizes for LOS and non-LOS.

There is no need to invoke the Space Tuner when the allocation speeds of both spaces have stabilized. Thus, we have implemented a simple function, *gc_decide_space_tune,* to check whether space tuning is necessary. This function first gathers information about the allocation speeds, the survivor ratios, and the size of free space of all partitions. Then from this information, it calculates the wasted space on the heap. For instance, if garbage collection is triggered by LOS, then the wasted space is the unused space in non-LOS at the moment when garbage collection is trigged. Next, it compares the wasted space to the threshold value. If the wasted space is greater than the threshold, then space tuning is necessary; otherwise, it eliminates the performance overhead of the invocation of the Space Tuner. Note that this threshold value can be

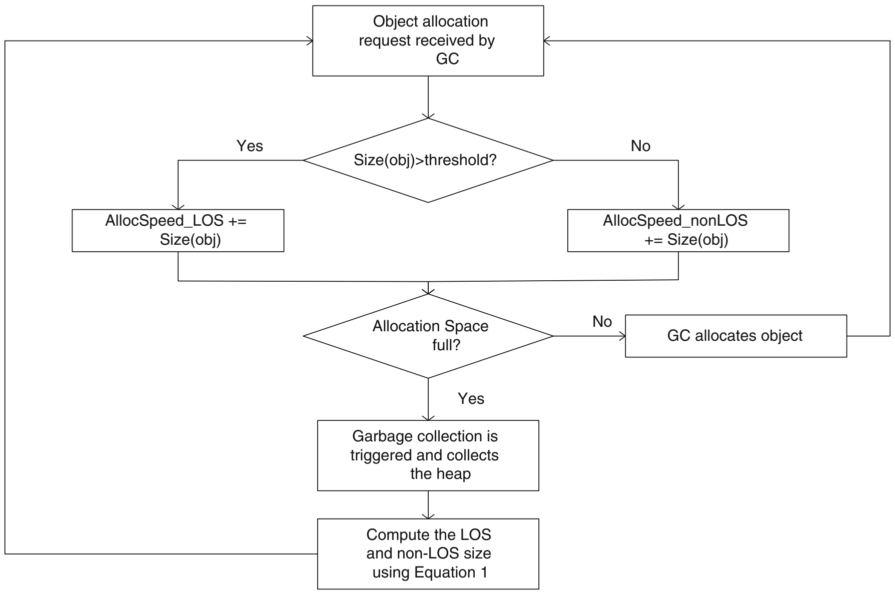
Object allocation request received by GC
Size(obj)>threshold?
Yes
No
AllocSpeed_LOS += Size(obj)
AllocSpeed_nonLOS += Size(obj)
Allocation Space full?
No
GC allocates object
Yes
Garbage collection is triggered and collects the heap
Compute the LOS and non-LOS size using Equation 1

Fig. 4 Space tuner algorithm

adjusted, by default it is set to be 4 MB in our design. With the Space Tuner, we expect that when a garbage collection happens, both LOS and non-LOS should be almost fully utilized, and as a consequence, the number of garbage collections should be significantly decreased compared to that of without the Space Tuner.

4.2 Parallel LOS Compaction

To demonstrate the effect of the parallel LOS compaction algorithm, we modified the Apache Harmony GC with separate allocation spaces such that the parallel Mark-Compact algorithm is utilized for non-LOS management, whereas our parallel LOS compaction algorithm is applied for LOS management. The two key steps in the parallel LOS compaction algorithm are the generation of dependency lists and the data movement. First, when the garbage collectors are trying to compute the new addresses of the objects, a number of disjoint dependence lists are generated. Figure 5 shows the pseudo-code for dependency lists generation: 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 of the large objects and inserts these blocks into the dependency lists (Label 3).

After the dependency lists have been constructed, the collectors can start moving the large objects in parallel. Each collector atomically grabs a dependency list and works on it independently. Thus, it only requires an atomic operation for each dependency list instead of for each block. The pseudo-code for this parallel large object compaction is shown in Fig. 6. 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

```
global_target_address = heap_start;
for (each collector thread in parallel ){
 Label1: // atomically grab a large object
    large_obj = pick_node_atomically(large_object_list);
    obj_size = num_of_blocks (large_obj) * size_of_block;
 Label2: //atomically increment global_target_adress with the object size aligned at block boundary
    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 list for the dependence between the source blocks and target blocks.
    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
```

Fig. 5 Generation of dependence lists

```
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
```

Fig. 6 Parallel large object compaction

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.

4.3 Parallel non-LOS Compaction

In order to evaluate the effect of our parallel compaction algorithm on non-LOS, we implemented a fully parallel LISP2 compactor in Apache Harmony. LISP2 compactor is one of the best-known GC algorithms thus we demonstrated the effectiveness of the proposed algorithms in LISP2. Note that the proposed algorithms are generic enough to be implemented in other GC designs that involve only two or three phases as well. Our design thus consists of following four phases for a collection:

Phase 1: *Live object marking.* It traces the heap from root set and marks all the live objects;
Phase 2: *Object relocating.* It computes the new address of every live object, and installs the value into the object header;
Phase 3: *Reference fixing.* It adjusts all the reference values in the live objects to point to the referenced objects' new locations;
Phase 4: *Object moving.* It copies the live objects to their new locations.

Although we parallelized all four phases, we only discuss the parallelization of phase 2 and 4, which are most related to our proposed design. The object relocating phase, or phase 2, computes the objects' target locations, without really moving the objects. In this phase, the dependency tree is constructed in a similar manner as the dependency list building process. The tree building process is bottom-up, from the leaf nodes to the root. To achieve this, source blocks are used for synchronization control such that collectors atomically grab source blocks in heap address order. That is, a compactor thread grabs a source block and a target block. For each live object in the source block, the collector computes its target address in the target block. When the target block has not enough space, the collector grabs the next target block. Then the collector adds the source block identifier into the target block's source block list, or dependency list. Since a source block in one list can be the target block in another list, at the end of this phase, the interconnections of the dependency lists form a dependency tree, which would be traversed in phase 4. When the source block has no more live objects, the collector grabs another source block 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.

In the object moving phase, or phase 4, the collectors traverse the dependency tree to move objects. The traversal process is top-down, from the root to the leaf nodes. When a collector has a sub-tree whose root block is ready to be filled with data, the collector simply traverses the sub-tree in a breadth-first order, copying data from child blocks to the parent block. The exact copying address of every object is installed in the object header as the forwarding pointer in phase 2. Once a root block finishes its filling, it is removed from the dependence tree, and its original child blocks become new roots. The algorithm continues recursively and finishes when reaching the leaf blocks. Any block that is the child of other block is not eligible as an initial root block. It is only picked up by the collector when all its parent blocks are removed from the dependence tree.

4.4 The Load Balance Algorithm

In the previous sections we have described parallel compaction algorithms for both LOS and non-LOS. These parallel compaction algorithms would achieve high performance if and only if the workload for each thread is balanced. But in our baseline design, load imbalance can occur. As shown in Fig. 7, if there are two threads working on large object 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.

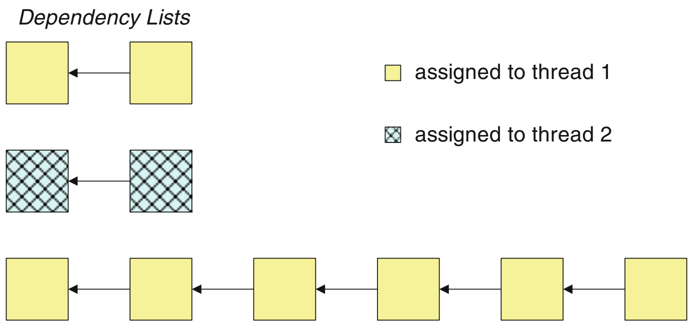

Fig. 7 Load imbalance in dependency lists

To maximize parallelism for high performance, 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/tree. 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/trees to count the number of blocks. However, the disadvantage of this design is that if the dependency lists/trees 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/trees are highly imbalanced. We can use a more sophisticated *Task Pushing* load balance approach [20]. For instance, when a dependency list gets too long or when there are always one or two disjoint sub-trees generated after the root block is filled, the work load between threads would be still 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 becomes a root block. Then after compaction finishes, this block in the reserved region (the virtual block) can be thrashed. This algorithm can be applied to dependency tree as well, but instead of breaking a long dependency list, we break a dependency tree into sub-trees and assign each sub-tree to a thread. The pseudo-code of this algorithm is shown in Fig. 8, where *Wi* is the local working set of collector

```
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 with virtual target block;
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
```

Fig. 8 Algorithm of load balance in parallel compaction

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.

5 Experiments and Results

We implemented the proposed algorithms in Apache Harmony. The evaluation of the design and implementation was done with SPECjbb2005 [17] and Dacapo [7] benchmark suites on an 8-core platform with Intel Core 2 2.8 GHz processors. Both benchmark suites represent data-intensive applications. Specifically, SPECjbb2005 is a large commercial server benchmark that allocates a large amount of non-LOS object. Thus it is suitable for the evaluation non-LOS garbage collection. On the other hand, the xalan benchmark in Dacapo is large-object-intensive, and it can be used for the evaluation of large object compaction. We used a 256 MB heap by default.

5.1 The Space Tuner

First we evaluated the performance of the Space Tuner. 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, xalan is an XSLT processor for transforming XML documents, it is large-object-intensive thus it requires a large LOS; on the other hand, SPECjbb2005 allocates a very small number of large objects, and it requires a large non-LOS. In addition, SPECjbb2005 actually 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. In this experiment, for SPECjbb2005 we ran eight warehouses, and for Dacapo we set size to large configuration.

As shown in Table 1, we compared three configurations, “tuner” denotes the utilization of our Space Tuner algorithm; “4M LOS” denotes the utilization of a static partitioning with 4 MB LOS; and “40M LOS” denotes the utilization of a static partitioning with 40 MB LOS. Column “num_collect” denotes the number of major collections triggered; and “wasted_frac” denotes the fraction of total heap space wasted when collection was triggered. The first observation from Table 1 is that static partitioning is highly inefficient. In several cases, including xalan-4M-LOS, luindex-4M-LOS, and chart-4M-LOS, the programs were not able to finish execution due to an out-of-memory exception, which was caused by inappropriate heap partitioning such that LOS is too small in these cases. With the Space Tuner, the space utilization became highly efficient, except in SPECjbb2005, all other benchmarks only required less than 5 major collections during their executions. Xalan allocated a lot of large objects, it required 1060 major collections even when LOS size was 40M, and on average, it wasted 53% of the total heap size when collection happened. Nevertheless, with the Space Tuner, major collection was only triggered once during its execution. For most of the Dacapo benchmarks, a 40M LOS was large enough to hold all the large objects, thus only one major collection was triggered in the cases of luindex, chart, and bloat.

Table 1 Performance of the Space Tuner

		num_collect	wasted_frac			num_collect	wasted_frac
SPECjbb	tuner	19	0.12	jython	Tuner	1	0.91
	4M LOS	43	0.3		4M LOS	467	0.74
	40M LOS	37	0.16		40M LOS	21	0.43
Bloat	tuner	1	0.91	luindex	tuner	1	0.96
	4M LOS	71	0.81		4M LOS*	1*	0.96*
	40M LOS	1	0.47		40M LOS	1	0.47
Chart	tuner	1	0.91	pmd	tuner	2	0.63
	4M LOS*	33*	0.91*		4M LOS	1170	0.86
	40M LOS	1	0.47		40M LOS	18	0.45
Hsqldb	tuner	4	0.325	xalan	tuner	1	0.91
	4M LOS	50	0.69		4M LOS*	44*	0.89*
	40M LOS	6	0.37		40M LOS	1060	0.53

The rows with a star * denote that the execution is not able to finish due to out-of-memory exception, meaning that the static partitioning is not suitable for the program behavior

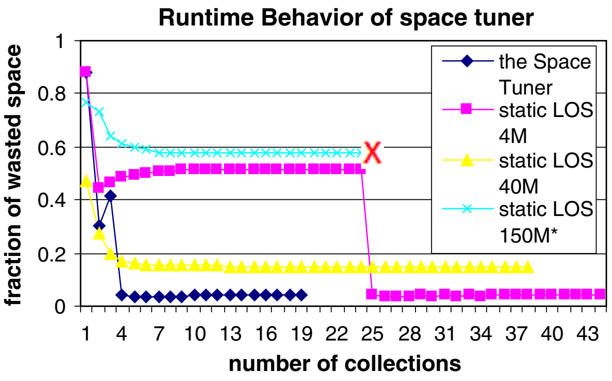

Fig. 9 Runtime behavior of the Space Tuner on SPECjbb2005

Note that the wasted fraction was high when the Space Tuner was used. Recall that the wasted fraction is defined as the fraction of heap space wasted when major collection happens. In most cases with the Space Tuner, major collection actually happened once. Thus, the high wasted fraction was actually caused by the improper heap partitioning set previously. When the Space Tuner detected this situation, it immediately adjusted the partitioning such that no further major collection was necessary.

Figure 9 shows the detailed runtime behavior of the Space Tuner running with SPECjbb2005. The *x*-axis shows the number of major collections and the *y*-axis shows the fraction of heap size wasted when collection happens. We compared the Space Tuner, static partitioning with a 4M LOS, static partitioning with a 40M LOS, and static partitioning with a 150M LOS. It clearly shows that with the Space Tuner, the heap partitioning stabilized after 4 collections. When further collections happened,

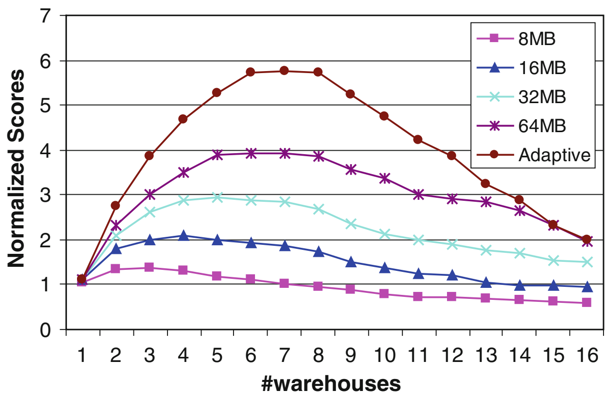

Fig. 10 SPECjbb2005 performance with different NOS sizes

the wasted fraction was less than 5%. On the other hand, the one with 150M static LOS failed to complete due to an out-of-memory exception, caused by insufficient non-LOS. Note that in the case of 4M static LOS, the wasted fraction decreased from 50% to about 5%. This demonstrated the dynamic object allocation behavior of SPECjbb2005: at the beginning of execution, SPECjbb2005 allocated a large amount of large objects, requiring a large LOS; later in execution, few large objects were allocated. The results in this section demonstrate that the Space Tuner greatly improved space utilization, and this optimization also brought time efficiency by reducing the number of collections.

Besides the LOS/non-LOS partitioning, we were also interested in how well the Space Tuner works with a generational copying collector. In this case we implemented a Space Tuner for NOS and MOS boundary adjustment and the results are shown in Fig. 10. In this case, NOS was collected by a copying collector, and MOS was collected by a compacting collector. The *x*-axis shows the number of warehouses used in the experiment, the *y*-axis shows the normalized score (a higher score implies higher performance), and we compared a 8, a 16, a 32, a 64 MB, and an adaptive NOS size. Figure 7 clearly demonstrates that NOS size adaptation with the Space Tuner can achieve much better performance than all of other fixed NOS size settings. Specifically, the performance is almost 5x better than that of the 8 MB NOS. This result demonstrates that the Space Tuner algorithm not only works for the space adjustment between LOS and non-LOS, but it can also be applied to any case that involves the partitioning of a space into multiple spaces.

In one of our recent experiments, we found that more accurate live object size estimation can be achieved by first estimating the survivor ratio of each block and then computing the total survivor size by adding up all survivor sizes in all blocks. Specifically, we can record the block survivor ratio in block headers or a specific block-indexed array, and then after collection is done we can get the live object size of the whole heap by adding up the live object sizes in all blocks. The key observation is that we can get more accurate estimation of the total survivor size by estimating the survivor size of each block. This is reasonable because as the granularity of estimation

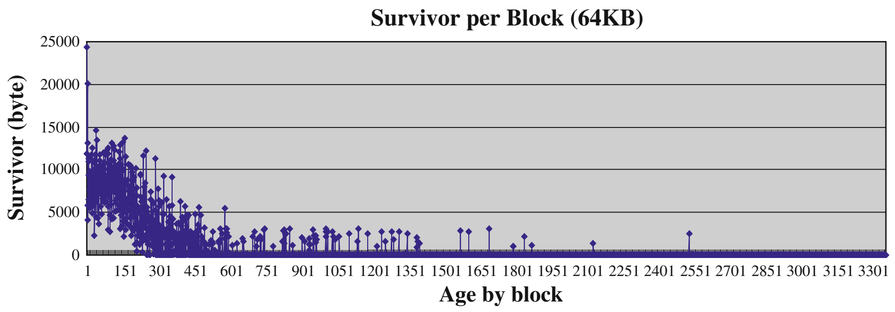

Fig. 11 Survivor ratio estimation at the block level

is reduced, e.g. from the heap level to block level, we utilize more detailed information about the allocation behavior and thus are able to make more accurate estimations. For instance, Fig. 11 shows the block-level survivor behavior of SPECjbb 2005, where the *x*-axis shows the block age counted as the number of collections, whereas the *y*-axis shows the survivor size in units of bytes. This Figure is an overlay of the survivor behaviors of all blocks in SPECjbb 2005, and it demonstrates that the survivor behavior is highly predictable. Furthermore, we found out that when the heap is not very large (e.g. 512 MB), the block-level survivor size estimation does not introduce a high overhead. Thus, the implication is that we may use block-level survivor estimation techniques in our Space Tuner to further improve space utilization. We will leave this optimization as our future work.

5.2 The non-LOS Parallel Compaction

In order to evaluate the effect of our parallel compaction algorithm on non-LOS, we implemented a fully parallel LISP2 compactor in Apache Harmony. In this experiment we ran SPECjbb2005 on an Intel 8-core machine. We first checked the scalability of this algorithm. We examined the time spent in different phases as shown in Fig. 12. The parallel compactor ran with 1, 2, 4, and 8 collectors. It clearly demonstrates that all four phases in our parallel LISP2 design achieved significant speedups. On average, the speedups of the four phases were 1.4x, 2.3x, and 3.7x respectively with 2, 4, 8 collectors. This result indicates that all four phases of our design are scalable. In other words, no individual phase would become the bottleneck of overall performance.

Figure 13 shows the impact of the parallel compaction algorithm on the overall GC performance. The metric we use here is GC pause time in units of milliseconds, which is the *y*-axis of Fig. 13. The results indicate that the overall normalized pause time had been reduced steadily from 100 to 70%, 43 and 27% of its original value as 1 thread, 2 threads, 4 threads, and 8 threads were used, respectively.

5.3 The LOS Parallel Compaction

To demonstrate the effect of the parallel LOS compaction algorithm, we modified the Apache Harmony GC with separate allocation spaces to incorporate this algorithm

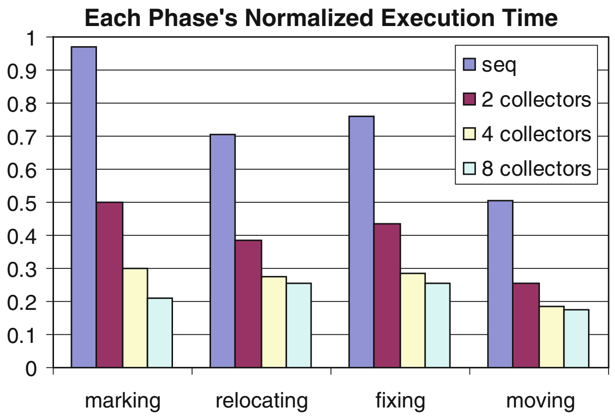

Fig. 12 Scalability of parallel non-LOS compaction

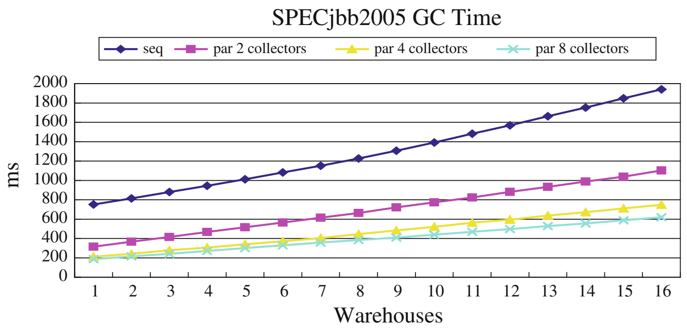

Fig. 13 The impact of the parallel compaction algorithm on the overall GC performance

for LOS compaction. We focused on the xalan benchmark because it is large-object-intensive. Figure 14 shows the scalability of this algorithm on xalan, pmd, and bloat. Besides xalan, the other two benchmarks are not large-object-intensive. We included them to demonstrate that the algorithm is still scalable even though the number of large objects is limited. The metric we use here is the normalized LOS compaction time. On average, the speedups of the parallel large object compaction were 1.56x, 2.10x, and 2.64x respectively with 2, 3, 4 collectors.

Then we studied how this parallel LOS compaction algorithm would impact the performance of the overall program execution. To get this data, we ran the respective benchmarks with 1, 2, 3, and 4 large object compaction threads and measured the total execution time. The results are organized in Fig. 15. It shows that when we ran xalan with 4 parallel compaction threads, a performance gain of about 3% was achieved. Although this seems to be a small performance gain, but since garbage collection only takes less than 10% of the total execution time, this result is actually a great improvement on GC performance. Note that other benchmarks, such as pmd, hsqldb, and bloat did not show significant performance gain or any trend of performance improvement.

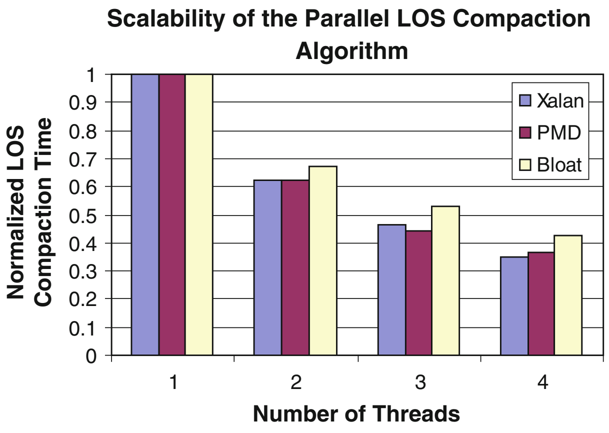

Fig. 14 Scalability of parallel LOS compaction

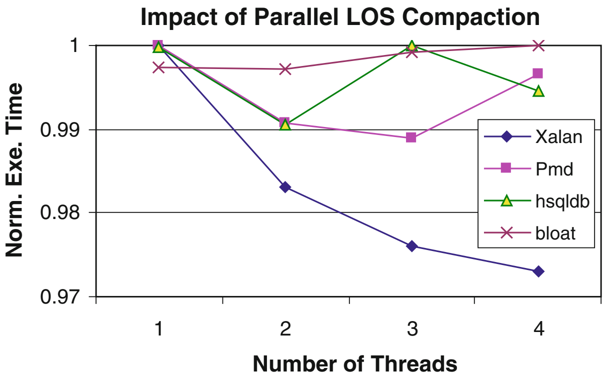

Fig. 15 Impact of parallel LOS compaction on overall execution time

This is because these benchmarks are not large-object-intensive, thus optimizations on large object compaction can induce little impact on the overall performance.

5.4 The Need for Load Balance

The proposed parallel compaction algorithms would achieve high performance only if the workload for each thread is balanced. 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 were highly 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 tree 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 trees that contained a large number of nodes, along with some small trees that contained only few nodes.

To address these problems, in Sect. 4.4 we have introduced two load balance algorithms *Task Collapse* and *Task Pushing*. *Task Collapse* introduces a low overhead but it does not guarantee perfect load balance, whereas *Task Pushing* introduces a higher overhead but meanwhile guaranteeing perfect load balance. In our experiments, we found that *Task Collapse* is sufficed for most benchmark programs. As demonstrated in Figs. 12 and 15, good speedups can be achieved with *Task Collapse* load balance techniques. Since *Task Pushing* needs to go through all dependency lists/trees, 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 ($\sim$ 10 GB). We will leave it as our future work to test the *Task Pushing* load balance algorithm in another setting, namely, high-end commercial servers with a large heap.

6 Conclusions

As multithreaded server applications prevail, Garbage collection (GC) technology has become essential in managed runtime systems. Space and time efficiency are the two most important design goals in garbage collector design. In this paper, we have proposed a complete algorithmic framework to improve both the space and time efficiency in parallel compacting GC design. This framework includes the Space Tuner, which dynamically adjusts the heap partitioning to maximize space utilization; the parallel compaction algorithm, which aims to fully parallelize the compaction process in both the large object space (LOS) and non-large object space (non-LOS) in order to achieve time efficiency.

We have evaluated the effectiveness of these mechanisms. The results show that the Space Tuner is able to largely improve the heap space utilization; this also leads to a certain performance improvement because the number of garbage collections has also been reduced. Further, we have demonstrated that our parallel compaction algorithm was scalable in both non-LOS and LOS. To test its effectiveness in non-LOS, we utilized this parallel compaction algorithm to produce a novel parallel version of the conventional LISP2 compactor and the results are encouraging. This algorithm offers an elegant and new solution to the well-known problem of parallel compaction in large object space.

Although we have proven our algorithms on a parallel compacting GC from Apache Harmony, these algorithms are by nature generic and have much broader applications. For the Space Tuner, we have demonstrated their effectiveness for both NOS/MOS boundary adjustment and for LOS/non-LOS boundary adjustment. It can indeed be extended to any design with multiple allocation spaces. For the parallel compaction algorithms, we have demonstrated their effectiveness in both LOS and non-LOS design. These algorithms can be extended to any design that involves object movements, such as those GC designs with two or three phases.

Our ongoing work is to apply the optimization techniques to the proposed algorithms. First, we plan to utilize block-size survivor size estimation in Space Tuner to further improve space utilization. Specifically, we need to evaluate the overhead introduced by this approach and determine under what situation it is suitable to utilize this optimization technique. 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*.

Acknowledgments This paper is an extended version of an earlier, smaller version that appeared in the Proceedings of ACM International Conference on Computing Frontiers 2009. This work is partly supported by the National Science Foundation under Grant No. CCF-0541403. 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. Abuaiadh, D., Ossia, Y., Petrank, E., Silbershtein, U.: An efficient parallel heap compaction algorithm. In the ACM Conference on Object-Oriented Systems, Languages and Applications, Vancouver, British Columbia, Canada (2004)
2. Apache Harmony: Open-Source Java SE. http://harmony.apache.org/
3. Azatchi, H., Levanoni, Y., Paz, H., Petrank, E.: An On-the-Fly Mark and Sweep Garbage Collector Based on Sliding Views. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (2003)
4. Barrett, D., Zorn, B.G.: Garbage collection using a dynamic threatening boundary. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, La Jolla, California (1995)
5. Caudill, P.J., Wirfs-Brock, A.: A third generation smalltalk-80 implementation. Conference proceedings on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, USA (1986)
6. Common Language Runtime Overview. http://msdn.microsoft.com/en-us/library/ddk909ch(vs.71).aspx
7. Dacapo Project: The Dacapo Benchmark Suite. http://www-ali.cs.umass.edu/Dacapo/index.html
8. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.: On-the-fly garbage collection: an exercise in cooperation. Commun. ACM **21**(11), 966–975 (1978)
9. Doligez, D., Gonthier, G.: Portable, unobtrusive garbage collection for multiprocessor systems. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1994)
10. Domani, T., Kolodner, E.K., Petrank, E.: A generational on-the-fly garbage collector for Java. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation
11. Endo, T., Taura, K., Yonezawa, A.: A scalable mark-sweep garbage collector on large-scale shared-memory machines. In: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing

12. Flood, C., Detlefs, D., Shavit, N., Zhang, C.: Parallel garbage collection for shared memory multiprocessors. In: Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium, Monterey, California (2001)
13. Hicks, M., Hornof, L., Moore, J.T., Nettles, S.M.: A study of large object spaces. In: Proceedings of International Symposium of Memory Management, Vancouver, British Columbia, Canada (1998)
14. Jones, R.E.: Garbage collection: algorithms for automatic dynamic memory management. Wiley, Chichester, July (1996). With a chapter on Distributed Garbage Collection by R. Lins
15. Kermany, H., Petrank, E.: The compressor: concurrent, incremental and parallel compaction. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Ottawa, Canada (2006)
16. Soman, S., Krintz, C., Bacon, D.F.: Dynamic selection of application-specific garbage collectors. In: Proceedings of International Symposium of Memory Management, Vancouver, British Columbia, Canada (2004)
17. Spec: The Standard Performance Evaluation Corporation. http://www.spec.org/
18. Steele, G.L.: Multiprocessing compactifying garbage collection. Commun. ACM **18**(9), 495–508 (1975)
19. Wegiel, M., Krintz, C.: The mapping collector: virtual memory support for generational, parallel, and concurrent compaction. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, Seattle, WA (2008)
20. Wu, M., Li, X.-F.: Task-pushing: a scalable parallel GC marking algorithm without synchronization operations. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium, Long Beach, California (2007)