ParMarkSplit: A Parallel Mark-Split Garbage Collector Based on a Lock-Free Skip-List

Nhan Nguyen[1] and Philippas Tsigas[1,*] and Håkan Sundell[2]

[1] Chalmers University of Technology, Gothenburg, Sweden
{nhann,tsigas}@chalmers.se
[2] University of Borås, Borås, Sweden
Hakan.Sundell@hb.se

Abstract. Mark-split is a garbage collection algorithm that combines advantages of both the mark-sweep and the copying collection algorithms. In this paper, we present a parallel mark-split garbage collector (GC). Our parallel design introduces and makes use of an efficient concurrency control mechanism for handling the list of free memory intervals. This mechanism is based on a lock-free skip-list design which supports an extended set of operations. Beside basic operations, it can perform a composite one that can search and remove and also insert two elements atomically. We have implemented the parallel mark-split GC in OpenJDK's HotSpot virtual machine. We experimentally evaluate our collector and compare it with the default concurrent mark-sweep GC in HotSpot, using the DaCapo benchmarks, on two contemporary multiprocessor systems; one has 12 Intel Nehalem cores with HyperThreading and the other has 48 AMD Bulldozer cores. The evaluation shows that our parallel mark-split keeps the characteristics of the sequential mark-split, that it performs better than the concurrent mark-sweep in applications that have low live/garbage ratio, and have live objects locating contiguously, therefore being marked consecutively. Our parallel mark-split performs significantly better than a trivial parallelization based on locks in terms of both collection time and scalability.

Keywords: garbage collector, concurrent programming, mark-split, mark-sweep, parallel garbage collection, lock-free data structures.

1 Introduction

Garbage collection (GC) is a form of automatic memory management to reclaim memory occupied by objects that are no longer used. Being introduced in 1960 [1], GC has evolved to become an important feature offered by many modern programming languages. Mark-sweep [1], copying [2], and their derivations are among the algorithms that have been extensively studied in the literature;

* The research leading to these results has been partially supported by the European Union Seventh Framework Programme (FP7/2007-2013) through the EXCESS Project (www.excess-project.eu) under grant agreement 611183.

M.K. Aguilera et al. (Eds.): OPODIS 2014, LNCS 8878, pp. 372–387, 2014.

and their pros and cons have been identified in a range of scenarios. The mark phase in mark-sweep has a time complexity proportional to the amount of live data, while the sweep phase has one proportional to the size of the heap. Mark-sweep can be improved by executing the sweep phase concurrently with the execution of the mutator, which has been suspended while marking. This technique is referred to as *lazy sweeping* [3]. Mark-region [4] improves the mark-sweep by dividing the heap in several regions and compacts objects to one end of the regions, and can thus reduce memory fragmentation. Garbage-First[5], which also works in per-region manner, marks objects and then evacuates them from current regions to new ones so that current regions can be reclaimed as a whole. Differing from the mark-sweep collectors, copying ones need time proportional to the amount of live data. However, they waste half of the space reserved for the need of the collectors, and move objects during collection. Copying collectors perform better than mark-sweep ones when the amount of live data is small compared to the size of the heap. This is the case where mark-sweep is penalized by the complexity of its sweep phase.

Sagonas and Wilhelmsson [6] introduced a GC technique called mark-split that can combine advantages of mark-sweep and copying collection. Mark-split evolves from mark-sweep but removes the sweep phase. Instead, the list of free spaces is built during marking, and can thus be used for allocation when the mark phase completes. Mark-split starts by creating the list of free intervals containing only a big free interval spanning the whole collected space. Then it proceeds to the mark phase. For each unmarked live object, it marks the object and calls a special *split* operation to exclude the marked space from the free intervals. The *split* operation which takes an object as an argument splits a free interval containing that object into two smaller free intervals, one to the left and the other to the right of the object. When the mark phase completes, the list of free intervals contains only free memory, thus can be used for new allocation.

Mark-split removes the sweep phase from mark-sweep, and thus achieves a time complexity proportional to the size of the live data set. However, this comes with an overhead cost of maintaining a set of free memory intervals. The number of free intervals is much smaller than the number of live objects because some live objects reside adjacent to each other. It seems beneficial, in certain situations, to avoid the sweep phase at the cost of this overhead, which depends on the distribution of live objects and also highly on the data structure selected to store the free intervals. The data structure should preferably provide search for an interval at sub-linear cost, e.g. binary search trees, splay trees, or skip-lists. The original mark-split uses a sequential balanced search tree [6], which might hurt its performance.

While mark-split is comparable to mark-sweep and even outperforms it in some situations, to the best of our knowledge, this is the first effort to design a mark-split collector for multi-core systems. Our contribution is to parallelize mark-split based on a highly concurrent data structure to handle the free intervals. We consider using lock-free data structures for their many advantages such as providing high performance, progress guarantees and immunity to deadlocks and livelocks [7]. However, previous implementations of concurrent data structures that supported the basic operations couldn't be used directly to parallelize mark-split, as they were not powerful enough to build a list of free intervals in mark-split. This was because mark-split frequently performs a combined operation of multiple basic operations. First it finds the correct interval and then performs *split* on it. This latter operation is also a combination of two operations; i) remove one interval and possibly ii) add two intervals. Concurrent environments require that *split* operations must perform all those actions in an atomic step, and thus the concurrency control is a challenge for the data structure to be used. A lock-free skip-list such as the one introduced in [8] can satisfy the performance but not the capability requirements of mark-split. We therefore extend it with a novel concurrency control to handle the free intervals and use the new skip-list to parallelize mark-split.

The rest of this paper is organized as follows. Section 2 introduces our extended skip-list algorithm to meet the requirements for parallelization of mark-split. The implementation of a parallel mark-split algorithm with the design of a lazy-splitting mechanism are presented in Section 3. Section 4 shows our evaluation of the GC inside HotSpot, along with result discussions before section 5 concludes the paper.

2 Concurrent Skip-List with Extended Functionality

We present a skip-list with extended functionality offering significant extensions over the original lock-free skip-list in [8]. A skip-list is a search data structure which stores elements in different layers of ordered linked lists with different densities to achieve tree-like behaviour. The original skip-list [8] can insert a new element, search for or remove an exiting element, but not a combination of those in one atomic operation. The use of recursion in that skip-list also made its memory management complicated and not efficient. Our extensions of the new skip-list are significant both when it comes to operations that it supports and in the algorithmic design. The new *replace2* operation gives the ability to atomically replace a node with one or two new nodes; making the skip-list usable in the context of mark-split. Regarding the performance, we redesigned the data structure to make use of hazard pointers[9] for memory reclamation purposes and thread-local-storage.

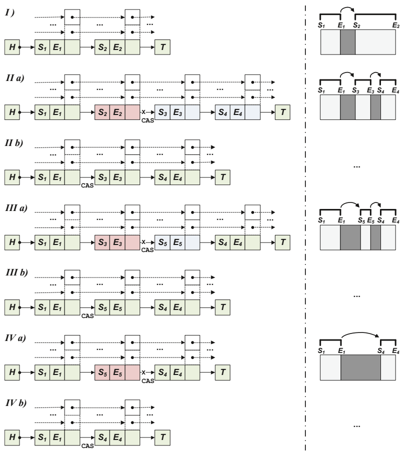

Fig. 1. Multiple-step process for marking and deleting blocks simultaneously with inserting new nodes, thus fulfilling the corresponding (to the right) abstract operations on the free-list

The *split* operation described in the mark-split algorithm operates on an abstract free-list representing a set of free intervals. A free interval can be represented by a node in a skip-list, where key represents the start address S of the interval and the corresponding value represents its end address E. As used in [8], the skip-list is basically made out of a singly-linked list with the nodes ordered by their keys. To allow probabilistic logarithmic expected time complexity for searching a particular node, nodes are inserted with a varying height such that several auxiliary lists are created with several layers of decreasing density with increasing height. For modifications to the abstract state of the free-list, only changes on the lowest layer's linked list are representative, i.e., changes are first performed atomically on the lowest layer and then modifications of the other layers can be performed concurrently with other operations. All necessary additional steps of the operation are eventually completed by making use of a suitably designed helping scheme. The helping scheme is designed to allow a concurrent operation to help another on-going operation when the former want to access the data that the latter is processing. A node in the skip-list can be defined to be *present* as soon as it is inserted on the lowest layer (i.e., there is another present node with a next pointer on the lowest level pointing to it) and *deleted* whenever the next pointer on the lowest layer for the corresponding node is marked (e.g. bit 0 set to 1). Atomic changes to the state of each node being present or deleted can be made using the Compare-And-Swap (*CAS*) primitive[1].

The *split* operation can result in four distinct changes on the abstract free-list. Each of these four changes must be possible to perform atomically with respect to each other. The possible changes are to either change S or E of an interval, replace the interval with two new intervals, or remove the interval altogether. To facilitate the representation of these abstract changes in the skip-list, an important observation is that it is possible to extend the skip-list to actually allow atomic deletion *and* insertion. The *CAS* primitive has the capability to both mark the next pointer and change it in the same operation. Thus, it is possible to atomically replace a node in the skip-list with one or more new nodes. The way that this modified skip-list is made to represent the abstract changes on the free-list, is shown in Fig. 1.

-**Step I** illustrates how a free-list containing the intervals $\langle S_1, E_1\rangle$ and $\langle S_2, E_2\rangle$ can be represented with two corresponding nodes in the skip-list.
- **In Step IIa,** the interval $\langle S_2, E_2\rangle$ is split into two intervals $\langle S_3, E_3\rangle$ and $\langle S_4, E_4\rangle$, where $S_3 = S_2$ and $E_4 = E_2$. By means of a *CAS*, the pointer on the lowest level of node $[S_2, E_2]$ is atomically marked and made to point to the new node $[S_3, E_3]$ which is already pointing to the new node $[S_4, E_4]$. The deleted node is then removed (also part of the helping scheme) in **step IIb**, with the *CAS* operating on the previous node's corresponding next pointer. The remaining layers are then handled in a similar manner.

[1] CAS, a synchronization primitive available in most modern processors, compares the content of a memory word to a given value and, only if they are the same, modifies the content of that word to a given new value.

- **In Step IIIa,** the interval $\langle S_3, E_3 \rangle$ is modified to become $\langle S_5, E_5 \rangle$ where either $S_5 = S_3$ or $E_5 = E_3$. By means of a *CAS*, the pointer on the lowest level of node $[S_3, E_3]$ is atomically marked and made to point to the new node $[S_5, E_5]$. The deleted node is then finally removed (also part of the helping scheme) in **step IIIb**, with the *CAS* operating on the previous node's corresponding next pointer. The remaining layers are then handled in a similar manner.
- **In Step IVa,** the interval $\langle S_5, E_5 \rangle$ is removed altogether from the free-list. By means of a *CAS*, the pointer on the lowest level of node $[S_5, E_5]$ is atomically marked. The deleted node is then finally removed (part of the helping scheme) in **step IVb**, with the *CAS* operating on the previous node's corresponding next pointer. The remaining layers are then handled in a similar manner.

The lock-free property is fulfilled by properly designing the helping scheme so that whenever an attempt made to perform a *CAS* for the a-part of the steps fails, the helping scheme makes sure that the b-part is being performed before attempting the a-part again.

2.1 Implementation

The implementation of the extended skip-list is described in Figs. 2 and 3 . The operation *split* removes a given interval (i.e., the *start* and *end* memory addresses of the live object) from the list of free intervals represented by the skip-list. The node that contains the given interval is searched for, with the search starting from the *head* node at the highest level. As the search is done in the skip-list level by level downwards, the previous node on each level is stored in the thread-local-storage *savedNodes* array. These remembered previous nodes are later used when deciding to either replace or remove the found node, according to the rules described in Section 2. If the found node, represented by *node*, is concurrently modified, the corresponding replace or remove attempts will fail, and the whole *split* operation is repeated.

Operation *replace2* describes how *node* can be atomically replaced by two new nodes *node*1 and *node*2. First the next pointer of *node* on the lowest level is atomically modified using *CAS*, to both contain the deletion mark (represented by the pointer value of 1) and instead point to *node*1. Thereafter, *node* is fully removed from the skip-list, and then *node*1 and *node*2 are inserted together, starting from level 1 and going upwards. During this insertion, *node*1 or *node*2 can have been concurrently deleted, in which case the insertion is aborted and helping is applied to make sure the deleted node is fully removed. Before actually starting modifying next pointers of previous nodes, the deletion mark is propagated upwards on all levels of the next pointer of *node* using *CAS* operations. This step is also required to be done by all concurrent operations that apply helping. The next step is then to modify the next pointer of all previous nodes of *node* such that they should instead point to the next node of *node*, starting with the highest level of the next pointers of *node* and going downwards. This is done by using *CAS* to atomically update the next pointer of the previous node, possibly given by $savedNodes[i]$, from originally pointing to *node* to instead

```
void split(void *start, void *end)
  do
    Node *node, *prev = head;
    for(i = MAX_HEIGHT; i >= 0; i--)
      for(;;)
        node = prev.next[i];
        if(node & 1)
          Go backwards in path using savedNodes[i + 1] or higher and help prev if needed
        if(node matches interval) break;
        prev = node;
      savedNodes[i] = prev;
    bool keepLeft = (start-node.start) ≥ T;
    bool keepRight = (node.end-end) ≥ T;
    int height = log2random(1, MAX_HEIGHT);
    if(keepLeft && keepRight)
      ok = replace2(node, new Node(node.start,start,height), new Node
          (end, node.end,height));
    else if(keepLeft)
      ok = replace1(node, new Node(node.start,start,height));
    else if(keepRight)
      ok = replace1(node, new Node(end,node.end,height));
    else
      ok = remove(node);
  while(!ok);

bool replace2(Node *node, Node *node1, Node *node2)
   Connect all next[] of node1 to node2
  do
    Node *next = node.next[0];
    if(next & 1) return false;
    node2.next = next;
    ok = CAS(&node.next[0], next, (node1 | 1));
  while(!ok);
  do_remove(node);
  for(i=1; i<node1.height; i++)
    do
      Node *prev = savedNodes[i];
      Node *next = prev.next[i];
       If prev is deleted or not the previous node according to node1, update savedNodes[i]
          while applying helping if necessary, and repeat
      node2.next[i] = next;
      ok = CAS(&prev.next[i], next, node1);
    while(!ok);
    If node1 or node2 has been marked for deletion, perform helping if needed and exit
        for-loop
  return true;

bool replace1(Node *node, Node *node1)
  do
    Node *next = node.next[0];
    if(next & 1) return false;
    node1.next = next;
    ok=CAS(&node.next[0], next, (node1 | 1));
  while(!ok);
  do_remove(node);
  for(i=1; i<node1.height; i++)
    do
      Node *prev = savedNodes[i], *next = prev.next[i];
       If prev is deleted or not the previous node according to node1, then update
          savedNodes[i] while applying helping if necessary, and repeat
      node1.next[i] = next;
      ok = CAS(&prev.next[i], next, node1);
    while(!ok);
     if node1 has been marked for deletion, perform helping if necessary and exit for-loop
  return true;
```

Fig. 2. Operations of the skip-list

```
struct Node
  void * start, end;
  int height;
  Node* next[height];
static Node *head = new Node(−∞, −∞, MAX_HEIGHT);
static Node *tail = new Node(∞, ∞, MAX_HEIGHT);
thread static savedNodes[MAX_HEIGHT];

void do_remove(Node *node)
   Mark node.next[x] on all levels x using CAS
  for(i = node.height−1; i >= 0; i−−;)
    Node *prev = savedNodes[i], *next = node.next[i] & (~1);
    bool ok=CAS(&prev.next[i], node, next);
    if(!ok)
      Update savedNodes[i] to be the previous node of node and perform helping if
          necessary of deleted nodes in the path, and repeat. If previous node cannot be
          found, perform next lap in the for-loop

bool remove(Node *node)
  do
    Node *next = node.next[0];
    if (next & 1) return false;
    ok = CAS(&node.next[0], next, (next | 1));
  while(!ok);
  do_remove(node);
  return true;
```

Fig. 3. Data structures, auxiliary *do_remove*, and *remove* operation of the skip-list

point to the next node. As concurrent helping can have been applied, it is important to notify this state when trying to update a possibly outdated pointer in $savedNodes[i]$.

Operation *replace1*, which replaces the free interval with another interval, follows the similar logic as *replace2* but only one new node, *node*1, atomically replaces *node*. *Remove* operation deletes a *node* as follow. First, the next pointer of *node* on lowest level is atomically modified using *CAS*, to contain the deletion mark. Thereafter, *node* is fully removed from the skip-list by *do_remove*.

For internal memory management, hazard pointers [9] are preferably used. Each hazard pointer represents a memory address that can be set by an individual thread in order to signal that the corresponding object is currently in use and should not be reclaimed. The thread-local-storage $savedNodes$ can then be implemented by a corresponding number of hazard pointers. To also allow the search part of *split* to safely pass through (i.e., de-reference) next pointers that are marked, without applying helping, the same hand-over trick as used in [10] can be applied.

2.2 Correctness

We now sketch (because of space constraints) the proof of correctness for the linearizability and lock-free criteria.

Lemma 1. *The implementation of the split operation, described in Fig.2, is linearizable with respect to other concurrent split operations.*

Proof: Linearizability is demonstrated by giving the respective linearizability points for the corresponding executions of the *split* operations in four cases:

Case 1 - split into two intervals: A *split* operation that results in this case takes effect at the successful *CAS* in line 30. Before the *CAS* takes effect, the nodes *node*1 and *node*2 cannot be reached by the search part of any concurrent *split* invocation, and *node* is not marked for deletion. After the *CAS* takes effect, the nodes *node*1 and *node*2 can clearly be reached by the search part of a concurrent *split*, as *node* is now referring to *node*1 as being the next node, and *node* has been logically deleted.

Case 2 - keep the left interval: a *split* that results in this case takes effect at the successful *CAS* in line 48 . Before the *CAS* takes effect, the node *node*1 (containing the left interval) cannot be reached by the search part of any concurrent *split*, and *node* is not marked for deletion. After the *CAS* takes effect, the node *node*1 can clearly be reached by the search part of a concurrent *split* as *node* is referring to *node*1 as being the next node, and *node* has been logically deleted.

Case 3 - keep the right interval: a *split* that results in this case takes effect at the successful *CAS* in line 48 . Same arguments holds as for Case 2.

Case 4 - remove the interval: A *split* that results in this case takes effect at the successful *CAS* in line 79 . Before the *CAS* takes effect, *node* is not marked for deletion. After the *CAS* takes effect, *node* has been logically deleted, which will be noted by any concurrent *split* operations that will fail to modify *node*, as the *CAS* in lines 30 and 48 requires the mark to not be set of the next pointer. □

Lemma 2. *The implementation of the* split *operation, described in Fig.2, is lock-free.*

Proof: The lock-free property of the *split* operation is maintained if a not finite execution of a loop for one invocation of the operation, is a result of a progress of another concurrent invocation. Assuming that the searched interval exists, the lines 6-10 are indefinitely repeated due to concurrent deletions. These deletions are due to successful concurrent *CAS* in lines 79, 30, and 48, all resulting in progress for the corresponding invocations. The lines 3-23 are repeated due to failed *replace2*, *replace1*, or *remove* functions. These functions fail in lines 28, 46, or 78, due to concurrent deletion of *node*. These deletions are due to successful concurrent *CAS* in lines 30, 48 and 79, all resulting in progress for the corresponding invocations. The lines 35-40 can indefinitely repeat due to concurrent deletions or insertions, which is progress for the corresponding invocations. Same arguments can be applied for the loops in lines 53-57 and lines 77-80. □

3 Parallel Mark-Split

We are first presenting the design of a lazy-splitting mechanism for our parallel mark-split algorithm, and then the main implementation, a.k.a ParMarkSplit.

3.1 Lazy Splitting

We design a *lazy-splitting* mechanism to improve the efficiency of the splitting part. Originally, whenever a live object is marked, an interval is split to exclude the space occupied by the marked object from the free intervals. We called this design *aggressive splitting*. Splitting for every marked object is inefficient in multi-threaded environment as it causes high contention at the shared data structure. We observe that: marking threads often consecutively mark objects that locate adjacent. The number of those adjacent marked objects is observed about 10% to 61% of the total number of live objects in applications in the DaCapo benchmarks. It is possible to perform splitting one time for adjacent objects that are marked consecutively, instead of splitting for each individual marked objects. We design a mechanism to do so, called *lazy-splitting*.

The lazy-splitting mechanism works as follows. Each marking thread maintains a memory range of adjacent objects recently marked but not yet "split". When it marks a new object, the object's memory is coalesced to the range if they are adjacent. Otherwise, it performs *split* for the range and the range is set to the object's memory. At the end of marking, *split* is called for the remained range. The lazy-splitting mechanism reduces the number of accesses by marking threads to the list of free intervals compared to aggressive splitting at a cost of maintaining not-yet-split interval locally at each marking thread. The lazy-splitting benefits the parallel mark-split algorithm when the performance gain by the reduction of the number of calls to *split* can cover the cost: $(N-M).C_1 > N.C_2$, where N is the total number of live objects; M is the total number of *split* operations that the lazy-splitting performs; C_1 is the cost of a *split* operation and C_2 is the cost to add a marked object to the not-yet-split interval. It is reasonable to assume that, for specific application and platform, these costs are constants. Therefore, whether the lazy-splitting mechanism benefits ParMarkSplit collector mainly depends on the $(N-M)/N$ ratio. An auto switch mechanism for determining when to use lazy-splitting is easy to design by using a threshold t to decide when to use lazy-splitting. Based on the evaluation results, we recommend $t = 10\%$. By default, lazy-splitting is applied as it is observed to benefit the parallel mark-split GC. But, lazy-splitting is not going to be applied when the GC finds, while collecting, that $(N-M)/N < t$.

3.2 Implementation

A parallel version of mark-split can be achieved by performing the following modifications to the concurrent mark-sweep collector (CMS) [11]

- When GC starts, empty the skip-list, then add one interval of the entire region to it.
- When a thread marks an object during the mark phases: If *aggressive splitting* is used, the thread calls *split* to remove the occupied space from the skip-list. If *lazy-splitting* is used, the thread book-keeps the object for the lazy-splitting mechanism.
- At the end of the Remark phase (i.e., the mutator is still suspended), convert the list of free intervals to the format of the allocator's free list. Remove the Sweep phase.

The correctness of the algorithm in the presence of interleaving among concurrent operations can be achieved thanks to the design of the extended skip-list which allows *split* to be performed atomically and in a lock-free manner. The lock-free property of the skip-list, when the number of objects to be marked is finite, guarantees the termination of all executed *split*, and therefore, the mark phases.

We implement our parallel mark-split collector as a collector for the 64-bit OpenJDK 7's HotSpot virtual machine - an open source implementation of the Java SE Platform contributed and supported by Oracle. The collector is named ParMarkSplit. The HotSpot uses a generational heap layout which divides its memory space into two parts: young and old generations. The young generation is to contain recently allocated objects, while objects that have been lived for a while are placed in the old generation. ParMarkSplit serves as a collector for the old generation, similar to CMS. One implementation issue of ParMarkSplit based on the CMS is that CMS is dedicated to work for the old generation in a generational heap. This brings difficulty for a plain comparison of the two algorithms in which they are used to collect a whole heap. Disabling the generational option in HotSpot so that the collectors work on the whole heap requires thorough modifications of the memory management that would have touched the HotSpot intensively. We find that it is more practical to maintain the generational heap layout, similar to other known commercial JVM, which also allows the comparison of the collectors in an industrial standard environment.

4 Evaluation

We are presenting an experimental evaluation of our parallel mark-split collector and comparing it with other collectors in the HotSpot, using the DaCapo benchmarks. Then we discuss the memory overhead and characterize applications that can benefit from ParMarkSplit. We opted to compare ParMarkSplit, with lazy-splitting (PMS) and without it (PMS1), to existing HotSpot's CMS as it was implemented based on CMS. Our evaluation also includes a lock-based parallel mark-split (PMS_Lock) which uses a binary search tree that relies on a single mutex lock to synchronizes concurrent accesses to store free intervals.

The collectors were evaluated in two scenarios. In the first scenario, GCs were configured to work in a stop-the-world mode where the mutator was suspended during collection. This setting allowed us to exclude the synchronization cost between the old GCs and the mutator. Such an execution provides a better look at the performance of the design itself. In the second scenario, the GCs were evaluated in the concurrent mode where the mutator was not suspended during collection. This is the scenario that CMS was designed for.

The DaCapo suite [12] was used for benchmarking. DaCapo contains a set of open-source, general-purpose JVM benchmarks, and is representative of real-world Java applications. We ran the benchmarks and reported results from representative applications which have rich memory behaviors, as tested by Gidra L. et.al [13] and Kalibera T. et al. [14]: *lusearch*, *avrora*, *sunflow*, *tomcat* and *xalan*. We also found that the Dacapo benchmarks use much less memory than our available memory and do not produce much garbage in the old generation. Too big heap might not trigger any old generation collection, though the young generation collection could be triggered often. In order to focus on garbage collection of the old generation, the heap sizes were chosen to be close to the benchmark's working set size. They were 50 megabytes (MB) for aurora, 400MB for xalan and 100MB for the others. Corresponding flags are set to allow the GCs working in multi-threaded mode. The other flags were left on default values.

The experiments were run on two contemporary NUMA multiprocessor platforms running Ubuntu Linux with kernel 3.0.0. One has two Intel Nehalem 6-core processors running at 2.4GHz with HyperThreading, 48GB of RAM, and support up to 24 concurrent hardware threads. The other has four AMD Bulldozer 12-core processors at 2.6GHz, 64GB of RAM and supports up to 48 concurrent hardware threads. In each experiment, we iterated a benchmark six times so that the old generation's collection can be triggered for several cycles.

4.1 Stop-the-world Scenario

In the stop-the-world scenario, we evaluate the lazy-splitting mechanism and the garbage collection time of the evaluated GCs in five applications of the DaCapo benchmarks. We varied the number of threads that collect garbage (GC threads). As we observed that the performance of the evaluated GCs does not change significantly above 15 threads (due to the known poor scalability of CMS), we report the results up to this number of GC threads.

We first evaluate the lazy-splitting mechanism by comparing the number of *split* operations performed by ParMarkSplit in each collection cycles before and after adopting lazy-splitting mechanism. In general, the lazy-splitting mechanism helps ParMarkSplit reducing the number of *split* operations to be performed. In avrora and sunflow, lazy-splitting can reduce this number by around 50%. But in xalan applications, the reduction is only about $4 - 6\%$. The reason may be that live objects marked by the GC in xalan interleave with garbage. Therefore lazy-splitting can not reduce the number of calls to *split* as much as in other applications. We expect that the lazy-splitting mechanism benefits PMS, in term of collection time, the most in avrora and sunflow.

The benefits of lazy-splitting are reflected in the performance of the ParMarkSplit collector. Fig. 4 presents the collection time of different GCs in the HotSpot in our Intel and AMD systems. In four out of five benchmarks on the Intel one, lazy-splitting helps reducing the collection time of ParMarkSplit, especially in avrora and sunflow. Only in xalan, the improvement of lazy-splitting are not clear as the gained performance is not enough to pay-off for the overhead cost. Comparing to PMS_Lock, the ParMarkSplit implementations perform significantly better in all applications. The performance of ParMarkSplit compared to CMS, however, are mixture of good and bad results. There are two applications, avrora and sunflow, in which ParMarkSplit works better than CMS. In others, CMS works better. In seeking for the reason of this result, we notice that avrora and sunflow have higher ratios of adjacent live objects over the total number of live objects compared to the other DaCapo applications. These applications can benefit ParMarkSplit from the caching effect as the GC accessing the same intervals for a short time and help ParMarkSplit to work more efficient. We analyze this observation further in section 4.4.

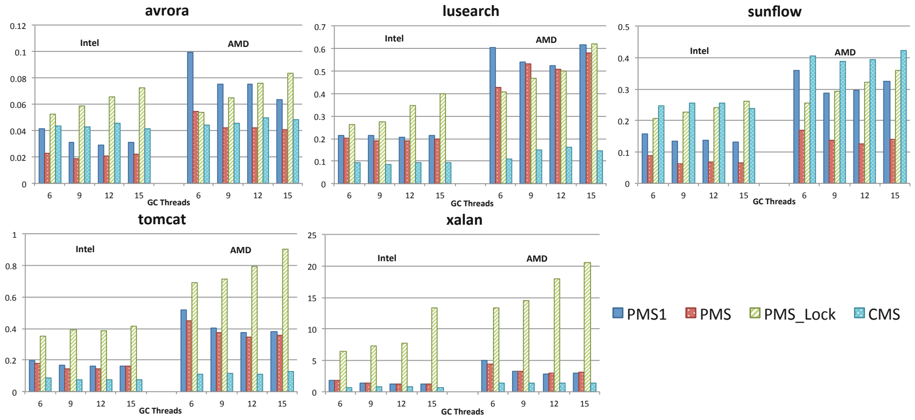

Fig. 4. Garbage collection time (sec) in the stop-the-world scenario

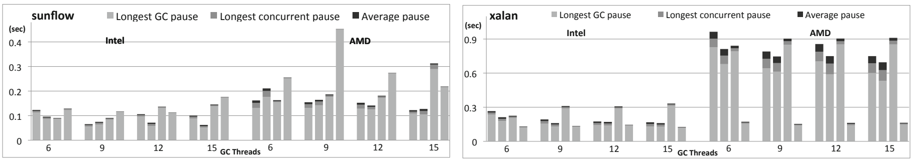

Fig. 5. Pause time when the GCs work concurrently with the mutator. Data columns at each label, from left to right: PMS1, PMS, PMS_Lock and CMS. *Longest concurrent (or GC) pause* when GC works in concurrent mode (or includes pauses when the GC switches to stop-the-world); *Average pause:* average of all the pauses by the old GC.

Another observation is the scalability of different GCs in Fig. 4. We can see that the PMS_Lock performs worse as the number of GC threads increases. This result is not surprised as the lock protecting the skip-list becomes a hot contention point when many GC threads concurrently access it. Meanwhile, the ParMarkSplit collectors, with and without lazy-splitting, as well as CMS are, at least, not scaling down its performance as the number of GC threads increases.

Considering the tested hardware platforms, we found that ParMarkSplit performs better on the Intel than on the AMD. One possible reason can be that the AMD system has NUMA architecture with four nodes which results in higher cost for accessing the shared skip-list.

4.2 Concurrent Scenario

In the concurrent scenario, GCs collect garbage concurrently with the mutator, i.e., the scenario that CMS was built for. We evaluated the pause times of our GC during the collection, in addition to the benchmark's execution times.

We measured the pause time at different number of GC threads. CMS suspends applications during the initial mark and remark phase. ParMarkSplit, which derives from CMS and adds the splitting work to these phases,

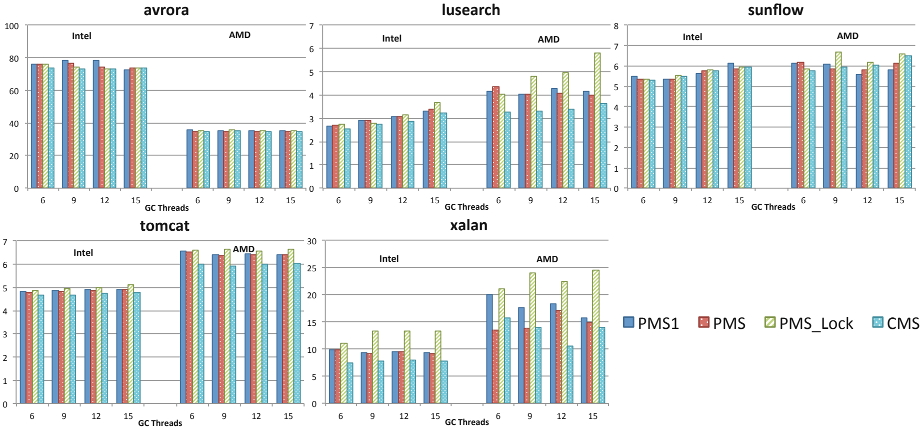

Fig. 6. Benchmark time (sec) for the HotSpot with different GCs

is expected to have longer pauses than the corresponding CMS's pauses. This reflects in the *longest concurrent pause* and *average pause*, which are pauses during concurrent collection, in Fig. 5. The same figure also shows the *longest GC pause* of the old generation GC which includes pauses when the collector switches to working in stop-the-world mode under certain circumstance, e.g the old generation is full during concurrent collection. Due to the lack of space, we include only the results of sunflow and xalan applications, representing for applications which may or may not benefit from ParMarkSplit. We can observe that both average and longest concurrent pauses of ParMarkSplit are longer than those of CMS, as expected from the design. In current HotSpot, the initial mark phase runs single-threaded while the remark phase, though can run multi-threaded, has many parts running sequentially as well. As these two phases run mostly sequentially, the pause time in ParMarkSplit, which uses lock-free synchronization based on compare-and-swap operation, are penalized dramatically. We can expect that when these two phases are fully parallelized in the HotSpot, pause time of ParMarkSplit will be improved significantly, at least proportionally to the speedup of the lock-free skip-list. Regarding the garbage collection pause time, we also notice that the longest GC pause time does not follow the trends of the longest concurrent pause time across the applications. In sunflow, the ParMarkSplit with or without lazy-splitting usually achieves shorter longest GC pauses than both the lock-based one and CMS. However, in xalan, the ParMarkSplit collectors have shorter longest GC pauses than the lock-based one, but longer than CMS. This observation can be drawn from both the AMD and Intel platforms. There are also different in term of absolute values between the two architecture. The AMD system usually has longer pauses than the Intel one.

Regarding the relation between the application's response time and the GC's pause time, it is noticeable that GC pause time is not necessarily the same as the application response time, which means how long it takes an application to responds to a request by users or by other applications. Even though pause time is an indicator for the maximum application response time in the worst case,

Table 1. The estimated size of the skip-list, and of the bitmap of Printesiz's technique

	avrora	lusearch	sunflow	tomcat	xalan
	Number of nodes (thousands) / Size (MB)				
Intel	2.0/ 0.3	14.4/ 2.1	4.9/ 0.7	49.1/ 7.1	55.0/ 7.9
AMD	2.2/ 0.3	16.6/ 2.3	4.3/ 0.7	46.4/ 6.7	57.6/ 8.3
	Estimated size of bitmap (MB)				
Bitmap	0.78	1.56	1.56	1.56	6.25

the contribution of the GC's pause time to the mean application response time is less and less important in systems with heavy loads, as studied by Persson M. and Cummins H. from IBM [15].

ParMarkSplit brings the split part to the mark phase, but it also removes the sweep phase. Does this change reflect in the overall throughput of the applications? Fig. 6 plots the execution time of the benchmarks at different numbers of GC threads. In some specific cases of lusearch and xalan applications, CMS performs better than PMS. In sunflow, however, PMS performs slightly better than CMS. Excepts for those cases, we did not observe significant differences in the benchmark's execution time between PMS and CMS. Comparing to the lock-based parallel mark-split, the benchmark times of ParMarkSplit are lower in most cases. ParMarkSplit has also shown that it works better than CMS in sunflow, both in terms of pause time and throughput. We will analyze the reason that ParMarkSplit works well in certain applications in section 4.4.

4.3 Memory Usage

We can estimate the memory overhead used to store free intervals based on the memory used by the skip-list. Each free interval is stored as a skip-list's node, which occupies approximately 18 memory words; two for the start and the end of the free interval, one for the node's level in the skip-list, and at most *max_level* pointers pointing to the next nodes in the linked list at each level of the skip-list. During its construction, the skip-list decides *max_level* so that 2^{max_level} is approximately its average size. As our estimated average number of free intervals is 32000, *max_level* is set statically to 15. The estimated memory used by the skip-list in a 64-bit system is presented in Table 1.

We observe that avrora and sunflow have the lowest number of free intervals among the benchmarks. This is because their marked live objects often reside next to each other as discussed above. The memory overhead in avrora and sunflow is less than 1% over the heap size (0.3/50MB and 0.7/100MB, respectively), which is negligible. This cost is higher in applications where the number of free intervals are high; approximately 2% in lusearch and xalan, and 7% in tomcat, where the heap sizes are 100MB, 400MB and 100MB respectively. The size of the skip-list is usually small in applications where their live objects often reside adjacent to each other, making the memory overhead become negligible. Compared to the memory overhead of Printezis's technique, which uses a bitmap to skip over contiguous unmarked objects while sweeping [11], ParMarkSplit uses less memory in avrora and sunflow, but more in other benchmarks.

The fragmentation behavior of ParMarkSplit is similar to that of CMS, as it is expected by design. It is possible to check the fragmentation level during or after a collection cycle by checking the size of the skip-list. When the heap is considered too fragmented, a compaction algorithm can be applied in a similar way as it is applied in the CMS garbage collector in HotSpot.

4.4 Characterization of Applications That Benefit from ParMarkSplit

We try to characterize the applications in which ParMarkSplit performs better than CMS so that the system can adaptively select the best GC based on these characteristics. We have observed from the experimental results that ParMarkSplit outperforms CMS in the sunflow and avrora applications, and not in tomcat and xalan. As ParMarkSplit performance is highly dependent on its most frequent operation, i.e., *split*, it usually performs better in applications where the number of live/garbage ratios are low. Analysis on those applications shows that sunflow and avrora have live/garbage ratios as low as about 15% and 20%, respectively. Tomcat have higher ratio; 40% on average. We can speculate that ParMarkSplit maintains the property of its sequential counter-part that it performs better in applications which have low live/garbage ratio. However, this property could not be applied to explain ParMarkSplit's performance in other applications with the same characteristics. Xalan have similar live/garbage ratios as sunflow and avrora, but ParMarkSplit does not perform well in them. We need to distinct the formers from the latters to better characterize the application that clearly benefit from ParMarkSplit.

We observed that our lazy-splitting design brings significant performance gains to ParMarkSplit in applications where it already performs better than CMS, i.e., sunflow and avrora. The benefit of the design in xalan is however not as much. The main characteristic differentiating the two groups is the ratio of the number of adjacent marked objects over that of total marked objects. This ratio is high in sunflow and avrora; and lower in xalan. When the ratio is high, doing splitting interval operation in ParMarkSplit benefits in two ways. First one is a cache benefit when a free interval that is previously split can be cached and reused in the next splitting. Second benefit is that only one *split* operation is required for a set of adjacent objects. As ParMarkSplit brings more such advantages to sunflow and avrora than to xalan, it performs better than CMS in the former applications but not in the latters in our experimental evaluation. All above observations regarding the characterization of applications that benefit from ParMarkSplit are consistent across the two evaluated hardware platforms.

To conclude, ParMarkSplit has been shown to perform better than CMS in applications where the ratio of the number of live objects to that of garbage objects is low and live objects often reside adjacent to each other. ParMarkSplit can be used as a complement to other garbage collection mechanisms to target applications with such characteristics.

5 Conclusion

We present a parallel design of the mark-split garbage collector, called ParMarkSplit. To the best of our knowledge, this is the first parallel mark-split design. The design is based on a lock-free data structure that extends the functionality of a skip-list to meet the requirements of the mark-split algorithm augmented with a lazy-splitting design. A complete implementation of the ParMarkSplit collector was developed and integrated in the OpenJDK HotSpot Java virtual machine. We evaluated its behavior experimentally and compared it with the default concurrent mark-sweep garbage collector present in HotSpot, using the DaCapo benchmarks. The experiments were performed on two multiprocessor systems of different architectures; Intel's Nehalem and AMD's Bulldozer. The results are encouraging in applications where the ratio of the number of live objects to that of garbage objects is low and live objects often reside adjacent to each other. We believe that ParMarkSplit can add weight to other garbage collection mechanisms when used for applications with such characteristics.

References

1. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. Commun. ACM 3, 184–195 (1960)
2. Cheney, C.J.: A nonrecursive list compacting algorithm. Commun. ACM 13, 677–678 (1970)
3. Hughes, R.J.M.: A semi-incremental garbage collection algorithm. Software: Practice and Experience 12(11), 1081–1082 (1982)
4. Blackburn, S.M., McKinley, K.S.: Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. SIGPLAN Not. 43(6), 22–32 (2008)
5. Detlefs, D., Flood, C., Heller, S., Printezis, T.: Garbage-first garbage collection. In: Proceedings of the 4th ISMM, pp. 37–48. ACM (2004)
6. Sagonas, K., Wilhelmsson, J.: Mark and split. In: Proceedings of the 5th International Symposium on Memory Management, ISMM 2006, pp. 29–39. ACM (2006)
7. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
8. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. 65(5), 609–627 (2005)
9. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Transactions on Parallel and Distributed Systems 15(8) (August 2004)
10. Sundell, H., Gidenstam, A., Papatriantafilou, M., Tsigas, P.: A Lock-Free Algorithm for Concurrent Bags. In: Proceedings of the 23rd ACM SPAA. ACM (2011)
11. Printezis, T., Detlefs, D.: A generational mostly-concurrent garbage collector. SIGPLAN Not. 36, 143–154 (2000)
12. Blackburn, S.M., et al.: The dacapo benchmarks: Java benchmarking development and analysis. SIGPLAN Not. 41, 169–190 (2006)
13. Gidra, L., Thomas, G., Sopena, J., Shapiro, M.: Assessing the scalability of garbage collectors on many cores. In: Proceedings of the 6th PLOS Workshop. ACM (2011)
14. Kalibera, T., et al.: A black-box approach to understanding concurrency in dacapo. In: The UK MM-NET Workshop on Memory Management (April 2012)
15. Persson, M., Cummins, H.: Java technology, ibm style: Garbage collection policies. IBM DeveloperWorks (May 2006)