Data Structure Aware Garbage Collector [*]

Nachshon Cohen
Technion
nachshonc@gmail.com

Erez Petrank
Technion
erez@cs.technion.ac.il

Abstract

Garbage collection may benefit greatly from knowledge about program behavior, but most managed languages do not provide means for the programmer to deliver such knowledge. In this work we propose a very simple interface that requires minor programmer effort and achieves substantial performance and scalability improvements. In particular, we focus on the common use of *data structures* or *collections* for organizing data on the heap. We let the program notify the collector which classes represent nodes of data structures and also when such nodes are being removed from their data structures. The data-structure aware (DSA) garbage collector uses this information to improve performance, locality, and load balancing. Experience shows that this interface requires a minor modification of the application. Measurements show that for some significant benchmarks this interface can dramatically reduce the time spent on garbage collection and also improve the overall program performance.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Dynamic Storage Management; D.3.4 [*Processors*]: Memory management (garbage collection)

General Terms Algorithms, Languages

Keywords Memory management, Data structures, Collections, Memory management interface, Parallel garbage collection.

1. Introduction

Garbage collection is a widely accepted method for reducing the development costs of applications. It is used in many of today's programming languages, such as Java and C#. However, garbage collection does not come without a cost, and automatic memory management adds a noticeable overhead on application performance.

Much effort has been invested in improving garbage collection efficiency, see for example [4–6, 10, 12, 16–18, 20, 23, 24, 30]. Most modern collectors employ a tracing procedure that discovers the set of objects reachable from a set of root objects. The tracing procedure is considered the most time consuming task, so most performance improvement efforts focus on it.

Knowledge about program behavior may greatly benefit tracing. Yet, today's programming languages do not provide interfaces to the application to pass on this information. Creating a good interface and garbage collector support for it is a challenge. First, we do not want to risk correctness even when the programmer provides bad hints. Recall that garbage collection was introduced to eliminate common bugs originating from programmer misunderstanding of program overall behavior. Second, a badly designed interface may require high programmer effort for producing relevant hints, and the programmer may not be willing to or not be capable of spending the time to collect such information. Finally, a naive interface design may end up not yielding improvements in garbage collection performance, and in this case there would be low incentive for implementing such an interface in the compiler or for using it even when implemented.

In this paper we propose the DSA (data structure aware) interface: an interface between the program and the memory manager that avoids the above pitfalls. First, the DSA interface allows the programmer to express the program's behavior, but in a way that requires very limited programming effort. Second, the information that is exposed to the garbage collector through the DSA interface does improve its efficiency and scalability, and also the overall program running times. And most important, a specification of incorrect program behavior through the DSA interface will never lead to program failure or inappropriate pointer handling, but only to decreased performance.

The DSA interface concentrates on data-management applications that are centered around data structures. A database server is a typical example: usually there exists a

[*] This work was supported by the Israeli Science Foundation grant No. 274/14 and by the Technion V.P.R. Fund.

ISMM'15, June 14, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3589-8/15/06…$15.00.
http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2754169.2754176Reprinted from ISMM'15, [Unknown Proceedings], June 14, 2015, Portland, OR, USA, pp. 1–14.

single data structure that represents a table (e.g., a balanced tree) and the database data is stored in these tables. Although there might exist several tables in the database, each table is typically an instance of the same tree. An example benchmark in this category is HSQLDB of the DaCapo benchmark suite 2006. Cache applications such as KittyCache also fall into this category. A cache application reduces accesses to a resource (e.g., the network or a database) by caching frequently accessed data. A typical cache implementation places all items in a single (large) hash table. An additional example of applications that can be improved by the proposed interface is the set of applications that deal with item management, e.g., a management application for a wholesale company, such as SPECjbb2005. Such an application tracks the number of items in each warehouse, the registered customers, and the orders being processed. A typical implementation would place items of the same kind in a data structure (a hash table or a tree). While the application may use several instances of such data structure, they typically all share the same main node class.

Applications of the above categories share two useful properties from a memory manager point of view. First, a large fraction of the objects are accessible only via a dominant data structure. Second, removing an item from the data structure is a well defined operation; the programmer actively selects an item and removes it. While it is not correct to assume that a removed item is unreachable and can be reclaimed, it *is* safe to assume that an object is *not* reclaimable before it is removed from a (reachable) data structure. This is the property that we exploit in the proposed interface.

On the other end of the DSA interface there is a DSA memory manager that benefits from the information passed by the program through the DSA interface. The DSA memory manager that we present allocates nodes of the data structure separately from the rest of the heap. This provides additional locality for the data structure. The DSA memory manager assumes that nodes in the data structure are alive unless they were declared as deleted. It uses this knowledge to improve garbage collection efficiency. Furthermore, placing all data structures together may improve locality of reference beyond just reducing the time spent on garbage collection. In fact, improved performance (beyond reduced collection times) is visible in all the programs we evaluated.

The developers may fail to report the removal of a node from the data structure. Such missing information about node removal is taken care of by an additional mechanism of the DSA that can reclaim such nodes. On the other hand, wrong hints provided by the program, i.e., notifying the collector that objects are deleted from the data structures while they are not, will never cause the garbage collector to reclaim a reachable object. However, it may cause the garbage collection to perform additional work and thus hurt performance, and temporarily increase the space overhead.

The DSA interface is not beneficial to all programs, as some may not use a major data structure to hold a significant fraction of its data. It is therefore important to ensure that a JVM using the interface does not claim a noticeable overhead of applications that do not use the interface. The design we propose only requires an overhead of one conditional statement in the class loader and a second conditional statement in the beginning of a GC cycle; evaluation shows (as expected) that the overhead in this case is negligible.

A typical use of data structures makes the application of the DSA interface even easier. Many programs use library implementations of data structures to organize their data. If the program chooses to use a library implementation that employs the DSA interface, then the developer needs to do very little to gain the DSA benefits. The library implementation will declare the relevant class as an appropriate data structure and will issue remove notifications to the interface. The one thing that the developer should take care of is to notify the interface when the entire data structure becomes unreachable, as discussed in Subsection 6.1.

We implemented the DSA memory manager in the JikesRVM [2] environment. We also modified several common data structures from the *java.util* package to support the DSA interface. We then evaluated the DSA memory manager on the KittyCache program [19], the pseudo SPECjbb2005 benchmark [27], the HSQLDB benchmark from the DaCapo suite 2006, and the JikesRVM itself.

For KittyCache [19], a garbage-collection-aware version required modifying 8 lines of the program code. These changes yielded a 40-45% improvement in GC time and a 4-20% improvement in overall running time (depending on the heap size and GC load). For the pseudo SPECjbb2005 benchmark, a garbage-collection-aware version required modifying four lines of code, and yielded a 24-28% improvement in GC time and a 2.8-6% improvement in the overall running time. For the HSQLDB benchmark, we modified three lines of code to obtain a 75-76% improvement in GC time, and a 31-32% improvement in overall running time. Finally, modifying the Jikes implementation to use a data-structure aware garbage collection required the modification of 8 lines of the Jikes code. These modifications were tested for runs of the DaCapo 2009 benchmark suite and yielded, on average, a 11-16% improvement in GC time and a 1-2% improvement in the overall running time. This experience shows that the DSA method is applicable for various kinds of applications and it may yield a significant improvement with a small modification effort, where applicable. We did not see a performance degradation in cases where the DSA was not applied.

Organization In Section 2 we provide some background and definitions. In Section 3 we present an overview of the DSA interface and the DSA memory manager. In Section 4 we define the DSA interface and the interaction between the application and the garbage collection. In Section 5 we describe the details of the DSA memory manager and discuss its benefits. In Section 6 we explain the requirements from the programmer. In Section 7 we describe an adaptation of the DSA algorithm for incorporating it into the implementation settings of the highly efficient Immix memory manager [6]. We present our performance results in Section 8, discuss some related work in Section 9, and conclude in Section 10.

2. Background and Preliminaries

2.1 Tracing Garbage Collectors

Many garbage collector algorithms employ at the heart of the collection procedure a tracing routine, which marks every object that is reachable from an initial set of roots. The tracing routine typically performs a graph traversal, employing a mark-stack which contains a set of objects that were visited but whose children have not yet been scanned. The tracing procedure repeatedly pops an object from the mark-stack, marks its children, and inserts each previously unmarked child to the mark-stack. The order by which objects are inserted and removed from the mark-stack determines the tracing order. The most common tracing orders are DFS (depth-first search) for a LIFO mark-stack, and BFS (breadth-first search) for a mark-stack that works as a FIFO queue.

While tracing is easy to understand and implement, tracing the whole heap efficiently is a more challenging task. In large applications, the heap contains hundreds of millions of live objects, spread over several GBs of memory. We now review some of the challenges in the current tracing procedures.

Work Distribution With the wide adoption of parallel platforms, the ability to utilize several processing units to execute a garbage collection has become acute. However, a good work distribution is not trivial to achieve. In practice, each thread typically uses a local mark-stack, and synchronizes with other threads by a work-stealing mechanism or when too much (or too little) work becomes available in its local mark-stack. While these heuristic methods provide reasonable performance in practice, they are not optimal, and their performance depend on the actual heap shape. While typical heap shapes provide a more-or-less reasonable scalability, there exist extreme cases that seem inherently hard to parallelize. The simplest example of a shape that is difficult to parallelize is a long linked list. Such difficult heap shapes do occur in practice [3, 13, 26]. The ability of our newly proposed collector to work with nodes of a data structure allows the tracing to be independent of the heap shape and creates an embarrassingly parallel tracing with excellent scalability.

Memory Efficiency: Locality It is well known that the locality of tracing tends to be bad, as the heap is traversed in a non-linear order. In addition, most objects are small, but the CPU brings from memory the entire cache-line they reside on. If two objects that share a single cache line are traced together, the pressure on memory is reduced, and efficiency increases. In the DSA memory manager, we allocate data structure nodes separately. This substantially improves the locality of the tracing procedure.

Mark-stack Management and Mark-stack Overflow The mark-stack that is used to guide the heap traversal poses a trade-off for a tracing run. A large mark-stack uses more cache lines, more TLB entries and implies more time spent on cache misses. On the other hand, a small mark-stack may cause a mark-stack overflow, which implies a large overhead. When the mark-stack overflows, it is possible to restore its state using an additional tracing over the heap. However, restoring the mark-stack is a very expensive operation, even if it happens just once in a collection cycle. Garner et al. [14] reported that mark-stack operations take 11%-13% percent of the trace. Dynamically allocated mark-stacks are sometimes used to handle such issues [4], but the management of these stack chunks has its own cost. The DSA memory manager reduces the use of mark-stacks in two ways. First, data structure nodes (that have not been removed) are not put in the mark-stack. Second, objects reachable from such data structure nodes are traced in a controlled manner in order to decrease mark-stack use.

2.2 Data Structure Nodes

Data structures are often implemented using *nodes*. These nodes are intended to abstract the data structure from the data itself, and allow data-structure algorithms to operate on abstract generic nodes, ignoring the specific data stored inside. Data is added to the data structure by allocating a new node that represents the data and inserting it into the data structure. Data is removed from the data structure by unlinking the node that represent the data from the data structure. Given this node-representation of data structures, the interface requires that the programmer specifies the classes used to represent data structure nodes and also specifies when the nodes are removed from the data structure. Many times a single node type (class) is used to represent the nodes of the data structure and so declaring a single class for the data structure suffices. Since a *remove* or a *delete* operation is typically a well-defined logical operation on nodes in the data structure, it is usually very easy to identify code locations in which a delete is executed. Again, many times there are very few such code locations that perform a delete and often the delete would happen in a single method of the data structure. These typical behaviors significantly simplify the programmer efforts for interacting with the DSA interface.

We emphasize the separation of data structure nodes and the stored data. In standard library data structure, the data structure node is a (private) internal class that is accessed only from within the data structure. The stored data, pointed to by data structure nodes, is used in an arbitrary (complex) manner by the program. For example, in the standard (library) data structure java.util.HashMap <Integer,String>, the data structure node is HashMap.Entry.

3. Overview

In this section we provide an overview of the DSA interface and the corresponding DSA memory manager. We start by defining the life cycle of a node. An instance of a *node* class is first *allocated*, and then *inserted* to the data structure. Many times these two operations occur in close proximity, usually in the same function. At the application's request, the node is *removed* from the data structure. At some later point, the node becomes unreachable by the application threads, and is then *freed* or *reclaimed* by the garbage collector.

The DSA interface lets the programmer specify which classes represent data structure nodes. For a data structure to be treated as such by the DSA memory manager, the interface must be used to annotate the class in order to announce it as a data-structure node class. Allocating an instance of an annotated class implicitly notifies the memory manager that the allocated object will be inserted into the data structure. A second part of the interface is a new garbage collector function, denoted *remove*, which lets the program notify the garbage collector that a node is being deleted from the data structure. When a node is removed from the data structure, the programmer must explicitly call the memory manager's *remove* function. As a consequence, the node will later be de-allocated during a garbage collection cycle, but only after the node becomes unreachable from the program roots. We stress that a *remove*d node may still be reachable by the application, even after being deleted from the data structure. In this case, it will not be erroneously reclaimed. Sometimes a node is allocated for insertion to the data structure but the insertion fails and the node is discarded before even being added to the data structure. The programmer should be aware of such a case, and this node needs to be passed to the memory manager's DSA interface *remove* function.

The DSA memory manager allocates the data structure nodes on specially designated pages, and tracks for each node whether it was passed to the *remove* function. During a garbage collection cycle, the garbage collector assumes that a non-removed node is still a member of a data structure and hence is still alive. Therefore, the garbage collector can treat the remaining set of non-removed data structure nodes as additional roots; these nodes are marked as alive and their children are traced.

During a garbage collection cycle, the tracing procedure benefits from this large set of objects that are known to be alive and are co-located in memory. Moreover, there is no dependency between the tracing of the nodes in this set. Thus, we let each thread grab (synchronously) a bunch of pages, and traces all live nodes that reside on these pages and add their descendants to the mark-stack. Pages are traced locally by a single thread and in memory order of nodes; that is, co-located nodes are traced consecutively.

Memory order tracing improves locality since co-located nodes are traced together. It also allows hardware prefetching, which further reduces memory latency. Multiple threads benefit from the lack of dependency, leading to good work distribution with lightweight synchronization. The "normal" roots are traced by an unmodified tracing procedure, and may exhibit bad work distribution. A thread that runs out of nodes to trace in the regular trace can then turn to tracing data structure nodes, which are always available for tracing. This provides an excellent load balance between threads that might suffer from idle times in a traditional garbage collection tracing.

4. Memory Management Interface

In this section we define the DSA interface between the application and the memory manager. The DSA interface includes one annotation and two functions that the memory manager provides. The annotation signifies that a class' objects are used as data structure nodes. The first interface function is used to let the memory manager know that a node has been removed from the data structure. The second interface function should be used infrequently. It asks the garbage collector to perform an extensive cleaning collection to identify and reclaim all nodes that have been removed from the data structure without proper notification from the program. We next name and explain each of these interfaces.

The annotation provided to the program is the *@DataStructureNodesClass* annotation, which is applicable for classes only. By annotating a class as *@DataStructureNodesClass*, every instance of this class is considered a node of a data structure and assumed alive until the application notifies the memory manager that the node is removed (via the first interface function). We denote a class that is annotated by the *@DataStructureNodesClass* annotation as an *annotated class*, and an instance of an annotated class is denoted an *annotated object*.

The first interface provided to the program is the *remove* function. It is the responsibility of the programmer to make sure that every annotated object is passed to the memory manager's *remove* function after it is deleted from the data structure, so that the garbage collector may free this object. Failing to do so would result in a temporary memory leak, and may slow down the application (though not result in memory failure). Unlike the free function in C, the program may still reference an object that was passed to the memory manager's *remove* function.

The parameter passed to *remove* must be an annotated object. Often this can be checked at compile time. Otherwise, a runtime check can be used to trigger an adequate exception. It is not necessary to limit the number of times that the program calls *remove* with the same object. An annotated object that was passed to the memory manager *remove* function is referred to as an object that was *announced as removed*.

A second interface function provided to the programmer is the *IdentifyLeaks()* function. This function is optional, and should be called infrequently or not at all. This function tells the collector that some objects may have been deleted from the data structure without a proper corresponding call to the collector's *remove* function. This may create memory leaks that this function is meant to solve (at a cost). A call to this function will make the next garbage collection cycle perform a full collection, which ignores knowledge about the data structures, finds and eliminates all such memory leaks.

The implementation of these functions is provided by the memory management subsystem of the virtual machine, and is described in Section 5.

5. The Memory Manager Algorithm

We now present the DSA memory manager algorithm that exploits the additional knowledge provided by the programmer via the DSA interface. The algorithm is presented as a modification over an existing memory manager algorithm, called the *basic* memory manager. For the current presentation we assume that the *basic* memory manager uses a mark sweep collection policy. Mark sweep collectors have a choice between keeping the mark bits inside the object or in a mark table aside from the objects. We assume that the *basic* collector places the mark bits in a side table which is a common choice for commercial collectors[4, 11], and fits well into our algorithm. In Section 7 we present a possible modification needed for running our algorithm on the high performance Immix collector, which places the mark-bits in the object header.

The dynamic class loader identifies classes annotated by the proposed *@DataStructureNodesClass* annotation. For each annotated class *A*, the class loader creates a memory manager for *A*'s objects. We call such a memory manager a *A memory manager* and we say that allocations of *A*'s objects are directed to the *A memory manager*.

The *A memory manager* holds a set of blocks (allocation caches) designated for the *A* objects. The *A* memory manager allocates on its designated blocks which are separate from the rest of the heap. Each such block is associated with a table denoted the *member-bit table*. The member-bit table has a bit for each object in the block, specifying whether the chunk is currently a member in the data structure. In fact this bit is set when the object is allocated and is reset to 0 when the *remove* function is invoked with this object. Note the linkage between the member-bit and the mark-bit that the garbage collector uses to mark objects reachable from the roots. When the member bit is on (and if the program properly reports membership to the collector through the interface) then we know that the object is reachable. On the other hand, when the member bit is not set, we know that it has been removed from the data structure, but it might still be reachable from the roots. Thus, having the member-bit set (and assuming proper use of the interface) implies that the mark-bit can be set at the beginning of the trace. We denote a block that is used to allocate annotated objects (of a data structure) as an *annotated block*.

ALGORITHM 1: Allocation

```
Output: Address of the newly allocated object of type A (DS
        node)
1  if allocation cache is empty then
2      A-allocator.getNonFullAllocationCache();
3      if all A allocation caches are full then
4          basic.getEmptyAllocationCache();
5          A-allocator.registerAllocationCache();
6      end
7  end
8  allocated = allocCache.alloc(); // same algorithm as
                basic but uses A allocation cache
9  memberBit(allocated) = true
10 return allocated
```

ALGORITHM 2: Remove

```
Input: Object object
1  if object type is not annotated then throw new Exception();
2  memberBit(object) = false
```

The *A memory manager* allocates objects just like the *basic* memory manager allocates objects. If *A*'s blocks are all full, an empty block is requested from the global pool of free blocks of the *basic* memory manager, and the allocation request is served from that block. Upon allocating an object, the *A* memory manager also sets the *member-bit* corresponding to the allocated object. The allocation procedure is presented in Algorithm 1.

During a call to *remove*, the memory manager clears the member-bit corresponding to the passed object. At that point the object may still be reachable as some threads may hold pointers to that node even when it is not in the data structure. The removal procedure is presented in Algorithm 2. The algorithm should be executed in an atomic manner, for example by using the CAS instruction.

The tracing procedure is presented in Algorithm 3. We now discuss it in detail. The first loop (Step 1) initializes the mark bits of data structure nodes (annotated objects) to their member bits. This means that an object that belongs to the data structure (and has its member bit set) is not scanned by the standard tracing procedure, since the trace ignores marked objects. We will give these objects a special treatment at the end of the trace in Step 7. This initial copying of the member bits into the mark bits is executed very efficiently as it can be performed at the word level rather than for each bit separately.

The barrier in Step 5 is used to prevent races between the unsynchronized access to the mark bits during the copying of the member bits at Step 2 and the synchronized access to

ALGORITHM 3: Tracing procedure

```
1  for each annotated block do
       // marks live annotated nodes
2      Copy member-bit table to mark-bit table (block)
3  end
4  roots locations = basic.collectRoots()
5  barrier(); // wait for other threads to finish
6  basic.trace(roots locations) // trace using basic GC
7  for each annotated block do
       // Trace annotated nodes
8      for each object in the annotated block do
9          if memberBit(object)==true then
10             Push the object's unmarked children to the local
               mark-stack
11         end
12         if mark-stack size reaches a predetermined bound
           then
13             Transitively trace local mark-stack.
14         end
15     end
16 end
```

the mark-bits at Step 6 during the marking of objects whose member bit are reset (announced as removed).

In Step 6 the trace procedure invokes the *basic* tracing procedure, which traces the root objects. A thread that exhausts its available work for this trace proceeds to work on data structure nodes (annotated objects) in Step 7. In Step 10 the children of each annotated object that has its member bit set are pushed to the local mark-stack. Once in a while, when the mark-stack gets filled beyond a predetermined bound, all objects in the local mark-stack are traced transitively.

The *basic* collector may provide parallelism for the original trace. However, parallelism can be limited by the heap shape or heuristics used. The scanning of the data structure nodes is embarrassingly parallel as we need to get their information from consecutive addresses in the memory. Work distribution for a multithreaded execution is done by dividing iterations of the for loop at Step 8 between threads. The granularity of work distribution can be as small as a single iteration per thread, or as large as several blocks per thread. Note that iterations can be executed in any order.

5.1 Handling missed *remove* operations

The above description assumes that the programmer never forgets to report a deleted object. In this section we discuss a simple solution to leaks of annotated objects whose removal is not properly reported. We call an annotated object *leaked* if it is not reachable from the roots, but has not been announced as removed via the *remove* procedure. To reclaim such objects we can simply invoke the regular garbage collector on the entire heap. The garbage collection identifies all unreachable objects whose member-bit is set. It then reclaims these objects and resets their member-bit. A simple implementation of this idea appears in Algorithm 4. This algorithm can be called when the garbage collection does not free enough space for required allocations, or following an explicit request by the application. The programmer may wish to call this procedure for example when a large data structure becomes unreachable and the programmer does not wish to announce the removal of each member object explicitly. Also, the programmer may invoke this procedure "once in a while" if the *remove* announcement are known to be inaccurate. It is also possible for the system to automatically invoke this procedure once every ℓ collection cycles, for an appropriate ℓ.

ALGORITHM 4: IdentifyLeak tracing procedure

```
1  roots locations = basic.collectRoots()
2  basic.trace(roots locations) //trace normally using basic GC
3  barrier(); //wait threads to finish tracing
4  for each annotated block do
5      Member-bit table = member-bit table AND mark-bit
           table // if(member-bit=true and
           mark-bit=false) then member-bit:=false
6  end
```

There is no need to modify other phases of the GC algorithm, e.g. sweep.

5.2 Performance Advantages over Standard Tracing

Next, we discuss the improvements of the DSA tracing algorithm over standard tracing algorithms that are not data-structure aware.

Parallel Tracing and Work Distribution The DSA collector can be easily parallelized with a good work distribution. The iterations in the loop of Step 8 can be executed in any order, and thus embarrassingly parallel. Furthermore, since the embarrassingly parallel Step 10 is executed after the standard tracing procedure, the excellent work distribution of Step 10 can aid a possible bad work distribution of the original algorithm. While some threads may be delayed by some unbalanced work, other threads need not wait, but can proceed with the data structure related work.

Disentanglement of Bad Heap Shapes The DSA garbage collector cleanly solves the complicated problem of data structures with deep heap shapes. Such data structures may harm parallel tracing [3, 13, 26], but if such structures are annotated using the DSA interface, then they can be parallelized even in the presence of many-cores. In fact, the existence of a large data structure with deep heap shapes will *improve* the scalability of the tracing, because imbalanced work distribution would be resolved by the starving threads switching to work on scanning the data structure.

Locality The tracing of data structure nodes in the DSA collector is executed in memory order rather than in an arbitrary order determined by object pointers in the heap.

Thus, locality of the tracing procedure and the ability of the hardware to automatically prefetch required data improve substantially. Two nodes that reside on the same cache line suffer only one cache miss, and TLB misses are similarly reduced.

Mark-stack Management and Control The mark-stack used in Step 6 of the DSA algorithm is expected to be smaller than the mark-stack used for the original trace. The (annotated) data structure nodes and objects reachable from them (only) will not appear in the mark-stack in the first stage of the algorithm in Step 6. Next, in the second stage at Step 7, the nodes inserted into the mark-stack are those that are reachable only from nodes of the data structure. Furthermore, when the mark-stack passes a predetermined limit, we trace the current list of objects before we add more to the stack. This is expected to provide smaller mark-stacks on average.

6. Programmer View

In this section we review the changes required in order for a programmer to use the DSA interface. We assume that the garbage collector in the runtime used supports the DSA interface. Given an existing program, the required changes are very lightweight. We separate the discussion to the design of data structures (which may be put in a library) and the use of data structures in a program. Let us start with the former.

6.1 Data Structure Designer

The designer of library data structures and also programmers of ad-hoc data structures can add the interface for data-structure-aware garbage collector with a minor effort.

First, the programmer has to annotate the data structure nodes by the *@DataStructureNodesClass* annotation. Second, the programmer has to call the interface *System.gc.DataStructureAware.remove(N)* method whenever a node *N* is removed from the data structure. We stress that calling the *System.gc.DataStructureAware.remove(N)* method does not free the object *N* but rather lets the garbage collector reclaim it later when the object becomes unreachable. Therefore one can call this method even if *N* is still used by the calling thread or other threads in the system.

Third, if the data structure contains a method for a massive delete of a set of nodes in the data structure or even the entire data structure, then every node should be passed to the memory manager's *remove* function, or alternatively the *IdentifyLeaks()* interface must be invoked.

Sometimes, the question whether a library data structure benefits from the DSA interface depends on the context. For example, a HashMap can be used in a context where it holds many items and is never entirely discarded, but it can also be used in another context where it holds a few items and is frequently discarded. The DSA interface is beneficial only in the former context. For such cases it is advisable for the designer of a library data structures to create two copies of each data structure: one with the DSA interface and one without. This allows users to use the garbage-collection-aware version of the data structure only when beneficial.

6.2 Using a Library Data Structure

Programmers often invoke standard data structures whose implementation is provided in a library. We now discuss the requirements of a programmer that uses a data structure that supports the DSA interface. The only requirement of such a programmer is that he will be aware of the deletion of entire data structures. Usually, nodes are inserted into and removed from the data structures. But once the data structure is not needed anymore, the program may unlink it and expect the garbage collection to reclaim all its nodes. This is the case that requires extra care for the DSA memory manager since such nodes will not be reclaimed. To solve this, the programmer may either actively invoke Algorithm 4 using the IdentifyLeaks interface to reclaim the unreachable data structure nodes, or the programmer can delete all nodes from the data structure before unlinking it from the program roots.

If the library provides both a DSA and a non-DSA versions of the data-structure, the programmer needs to decide which version to use. A programmer may use both versions simultaneously in the same program.

6.3 Some Experience with Standard Programs

To check how difficult and effective the use of the DSA interface is, we have modified KittyCache[19], pjbb2005[27], the HSQLDB benchmark from the DaCapo test suite [7], and the JikesRVM Java virtual machine[2]. We are not authors of the these programs; nevertheless, we discovered that the required modifications were easy and did not require a deep acquaintance with the programs involved.

In all our attempts to use the DSA interface, we only used it when the program had a substantial fraction of its data kept in a small number of data structures. We did not attempt to use the DSA with a large number of small data structures, and we believe that the overhead and fragmentation that such use will create will make it non-beneficial. Note that in general, when the use of DSA turns out to be non-beneficial for any application, it is possible to remove all overhead by simply not using it for that application.

KittyCache The KittyCache is a simple, yet efficient, cache implementation for Java. It uses the standard library ConcurrentHashMap for the cache data, and the standard library ConcurrentQueue to implement the FIFO behavior of the cache. In order to make KittyCache use GC-aware data structures we only needed to modify the library code of these two data structures and nothing in the KittyCache application itself. The changes to the library data structures are described in Subsection 6.4 below.

pjbb2005 The SPECjbb2005 benchmark emulates a wholesale company, and saves the warehouses data in a standard HashMap. Again, to gain the advantages of the DSA interface, we modified the library code for the HashMap data structure.

The pseudo SPECjbb2005 (pjbb2005) is a small modification for the research community, which fixes the number of warehouses and measures running time for a specific workload instead of throughput. Additionally, it runs the benchmark multiple times to warm up the JIT compiler and measure only the last iteration. After each iteration, the data structure is discarded without freeing its entries. For this application one more modification was required to make it work with the DSA interface. To avoid going over the data structures and applying a remove notification for each node, we added a call to *IdentifyLeaks* at the end of each such iteration. This amounts to adding a single line of code to the benchmark itself. (A similar line would be needed also if we were using the original SPECjbb2005 benchmark. It would deal with entire data structure deletions at the end of each warmup phase.)

HSQLDB The HSQLDB is a SQL database written in Java. The HSQLDB benchmark in the DaCapo 2006 suite tests the database performance. Most of the database data is stored in a tree-like structure, a custom (and rather complex) data structure. Yet, adding garbage collection awareness was simplified by the fact that the main data structure node has a delete function which is called by the original code in an orderly fashion whenever a node is deleted. We annotated the data structure node class, and called the memory manager remove inside the (already existing) database delete function. Only two lines were added to the database code.

Similarly to pjbb2005, the entire data structure is discarded in each iteration. Thus, we used the *IdentifyLeaks* interface before each iteration started.

The newer DaCapo 2009 test suite contains another benchmark that tests the performance of a different SQL database called h2. The h2 database uses a different tree-like structure to store its data. Still, it already contains a delete function, so applying the DSA interface to this database was as easy as applying it to the HSQLDB database. Although we could easily modify this benchmark to be GC-aware, we ended up not measuring its performance because the Jikes RVM could not execute this benchmark.

JikesRVM The JikesRVM uses a library version of LinkedHashMap for organizing the ZipEntries. The LinkedHashMap forms a substantial data structure for Jikes RVM. This data structure is used (to the best of our understanding) to handle entries in the executed Jar file. We modified the library implementation of LinkedHashMap to be GC-aware. Library modifications are discussed in Section 6.4.

In addition, for the JikesRVM, it turned out to be useful to also annotate the ZipEntry objects themselves (the objects that the hash nodes reference). These nodes are not linked in a data structure of their own, yet, they are deleted only when their parent (a hash node) is deleted and so there is a well defined point in the program when such a node gets unlinked from the data structure that references it. These nodes typically become unreachable shortly after their parent node in the hash data structure gets deleted. We had to identify a couple of places where a node is allocated and not inserted to the data structure (e.g. cloning a node). For these allocation points we cleared the member bit immediately after allocating the node. This was slightly more complex (but still only a few hours of work), and required the modification of 4 lines of code in Jikes RVM.

6.4 Some Experience with Standard Data Structures

For data structure examples, we have transformed several data structures from the classpath GNU project (version 0.97.2) to fit the DSA interface. The changes for HashMap include three lines of code. The modified code can be shortly presented in a diff-like syntax.

```
+@DataStructureNodesClass
static class HashEntry<K, V> extends

public V remove(Object key){
  if(equals(key, e.key)){
+  System.gc.DataStructureAware.remove(e);

public void clear(){
  Arrays.fill(buckets, null);
  size=0;
+ System.gc.DataStructureAware.IdentifyLeaks();
```

The changes to the LinkedHashMap, TreeMap and LinkedList are very similar, and we do not present them here. It was possible to use a simple inspection of method names in order to determine the code locations where nodes are removed. In the TreeMap data structure, nodes are removed in a dedicated function, which makes the modification even simpler.

We also modified the ConcurrentHashMap and ConcurrentQueue from the classpath GNU (version 0.99.1-pre). Special care was needed in the rehash function of ConcurrentHashMap since objects are cloned. In the ConcurrentQueue data structure, special care was taken to call remove on the data structure sentinel. These changes took us few hours to identify without prior acquaintance with the data structure, and we assume that a programmer of a custom data structure will be able to identify these places even more easily.

In all our modifications of a library data-structures, we generated a copy of the original data structure implementation and added the DSA interface to the copy. We then used the copy whenever we needed to use the interface.

6.5 Shortcoming of our algorithm

For some programs we were not able to improve performance using our interface. We note two main reasons for that.

1. The program has no dominant data structure. This is the case for most of the DaCapo benchmarks.
2. The program uses a custom data structure without a clear interface (no remove function). An example to the above is the PMD benchmark in the DaCapo suite 2009. There exists a dominant data structure, whose nodes are objects of the *pmd.ast.Token* class. However, its data-structure pointers are public, and are modified in many code locations. We cannot tell if the programmers of PMD have invariants in mind that help identify code locations in which a node is removed, or whether they would also find it difficult to identify these locations.

7. Adaptation for the Immix Memory Manager

We have implemented the DSA interface and algorithm on top of the JikesRVM [2] environment version 3.1.3. Initially, we used the *basic* memory manager that uses a simple mark sweep garbage collector. However, although the DSA mechanism achieved significant improvement, this memory manager runs very slowly compared to other high performance memory managers. So, we proceeded and implemented our algorithm over the high performance Immix[6] memory manager. This required several modifications that we describe below.

The main issue with the Immix memory manager is that it puts the mark bit in the header of each object and not in an auxiliary table. This complicates the implementation of the DSA algorithm. When the DSA algorithm traces a data structure node, it needs to check whether its children are marked or not. If the mark bit is located in the children headers, the tracing would incur the cost of cache misses over accessing the children, even if both are members of the data structure. Even on a singly-linked list, each child is accessed at least twice, possibly by different threads. Thus, a naive implementation would nullify the memory locality benefit of our algorithm.

A possible solution is to identify the children which are data structure objects, and ignore them during the trace. Intuitively this seems correct since nodes are deleted from the data structure by unlinking data structure pointers to them. Thus, pointers from within the data structure to deleted nodes should not exist. However, ignoring children during trace may lead to correctness problems, especially when relying on an untrusted programmer. Instead, in the case of a child that is an annotated object we first check its member bit. If the member bit is on, there is no need to further trace the object, since it will be traced by the DSA tracing. Since the member bits are placed in a side table, checking the member bit usually hits the cache, instead of taking the cache miss over the child.

Another problem is marking all annotated objects prior to root tracing. When the mark bit is placed in the object header, marking all objects requires loading all of them to memory. Therefore, we modify the algorithm in the following way. The algorithm first traces all annotated objects, and then continues with the root tracing. This provides an initial embarrassingly parallel tracing for the data structure node but tracing other nodes on the heap does not get a balancing guarantee. The tracing function, for the case that the mark bit resides in the object header, is presented in Algorithm 5.

ALGORITHM 5: Tracing procedure for Immix

```
roots locations = Jikes.collectRoots()
barrier(); // same as immix
for each annotated block do
    for each object in the annotated block do
        if memberBit(object)==true then
            Mark object
            for each data structure pointer P of the object do
                if memberBit(P)==false then
                    ProcessChild(P) // push to
                        markstack if not marked
                end
            end
            for each non-data structure pointer P of the
            object do
                ProcessChild(P)
            end
        end
        if the mark-stack size reaches a predetermined
        bound then
            Transitively trace local mark-stack.
        end
    end
end
Jikes.trace(roots locations) // trace using Immix trace
```

Finally, the *IdentifyLeak* tracing procedure cannot go over the side tables (mark bit and member bit) to fix unreachable member nodes. Instead, it needs to go over the nodes themselves and check whether the nodes are alive or not. Since the IdentifyLeak tracing procedure is not called frequently, this modification should not affect the algorithm performance significantly. In our implementation we set the triggering of the *IdentifyLeak* tracing procedure to include only specific invocation of the IdentifyLeaks interface, but not ”once-in-a-while” invocation.

An alternative that we did not use is to trace the annotated objects after the Immix tracing loop, but modify the trace to not scan data-structure nodes with a set member bit. The problem is that this adds some performance overhead to the core of the tracing loop and so may reduce performance.

In Algorithm 5 Step 16 the local mark-stack is traced after a predetermined bound, which is a parameter of the algorithm. The value of this parameter represents a tradeoff. A lower value decreases the size of the mark-stack, which is good, but it also reduces the locality advantage for tracing consecutive objects, which is not good. In our implementation the mark-stack is traced after scanning all DSA objects that correspond to a single word in the member-bit table.

8. Implementation and Evaluation

We compared the DSA collector with the original unmodified Immix memory manager [6]. For the DSA memory manager, the data structure nodes and the member-bit table are part of the heap and are accounted for heap space usage, so the comparison is fair. The measurements were run on two platforms. The first features a single Intel i7 2.2Ghz quad core processor (HyperThreading enabled), an L1 cache (per core) of 32 KB, an L2 cache (per core) of 256 KB, an L3 shared cache of 6 MB, and 4GB RAM. The system ran OS-X 10.9 (Mavericks). The second platform featured 4 AMD Opteron(TM) 6272 2.1GHz processors, each with 16 cores, an L1 cache (per core) of 16K, L2 cache (per core) of 2MB, an L3 cache of 6MB per processor, and 128GB RAM. The machine runs Linux Ubuntu with kernel 3.13.0-36.

For each benchmark we measured performance of the modified benchmark with the DSA memory manager, and the unmodified benchmark with the original Immix collector. For each benchmark we report measurements on different heap sizes. We invoked each benchmark 10 times (10 invocations), and reported the average (arithmetic mean) of their running time and the average of the garbage collection time; the error bars reports 95% confidence interval. To reduce overhead due to the JIT compiler, in each invocation we measured only the 6th iteration, after 5 warm-up executions.

The first benchmark we measured was the HSQLDB benchmark from the DaCapo 2006 suite. The HSQLDB is a SQL database written in Java, and the HSQLDB benchmark test its performance. The changes to the benchmark code were discussed at Section 6. The minimal heap size was 130MB. We ran the program with 1.2x, 1.5x, and 2x heap sizes. The benchmark calls *System.gc()* manually, so garbage collection is invoked before the heap is exhausted and so the performance differences between different heap sizes were negligible. The total running time and total GC time are presented in figure 1. The DSA version improved GC time by 75-76% and overall time by 31-32%.

Next, we present the measurements of the DSA garbage collector behavior with the pseudo SPECjbb2005 benchmark. The pjbb2005 benchmark models a wholesale company with warehouses that serve user's operations. The warehouse data is mostly stored in a HashMap, which we modified in the DSA algorithm; the changes to the benchmark code were discussed at Section 6. We ran the benchmark with 8 warehouses, and fixed 50,000 operations per warehouse. The minimal heap size was 500MB; we ran the program with 1.2x, 1.5x and 2x heap sizes. The total running time and the total GC time are presented in figure 2. The DSA version improved GC time by 24-28% and overall time by 2.8-6%.

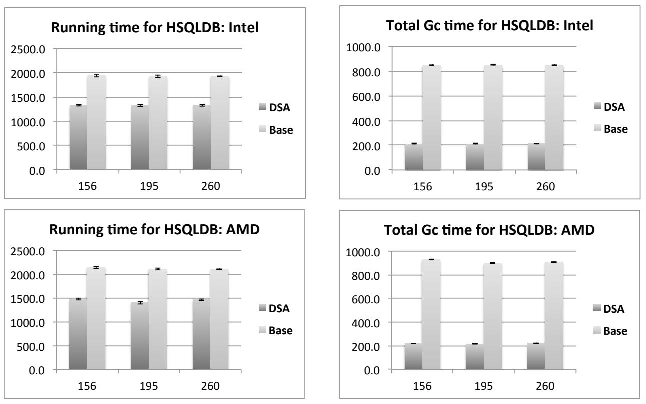


Figure 1. Total running time and total GC time (in milliseconds) for *HSQLDB* benchmark. The *x*-axis is the heap size used.

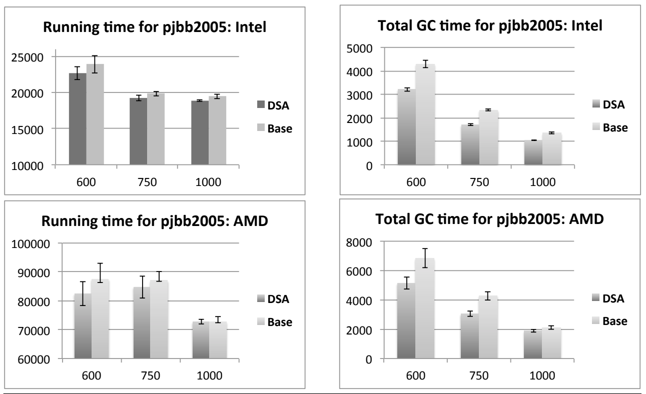


Figure 2. Total running time and total gc time (in milliseconds) for the pjbb2005 benchmark with 8 warehouses and 50,000 transactions per warehouse. The *x*-axis is the heap size used.

We then measured the DSA garbage collector with the KittyCache stress test. It allocates a cache of size 250K entries, and then executes 2,000,000 put commands. We ran the cache with different heap sizes; the minimal heap size was 200MB. The total running time and total GC time for various heap size are presented in figure 3. The DSA version improves GC time by 40-45% and overall time by 4-20%.

For these benchmarks we estimated the mark-stack usage by the baseline and the DSA versions. Accounting the mark-stack exact size requires extensive synchronization. Instead, we noticed that each thread has a local mark-stack and there is a shared pool of local mark-stacks when the local mark-stack is exhausted (denoted work-packets). Thus we measured the number of work-packets in the shared pool. Each work-packet (or local mark-stack) is exactly a single 4K page. While this is not an exact evaluation, it provides the amount of memory reserved for the mark-stack usage and approximates the mark-stack usage. We executed each benchmark 3 times with 1.5x heap size and record the max-

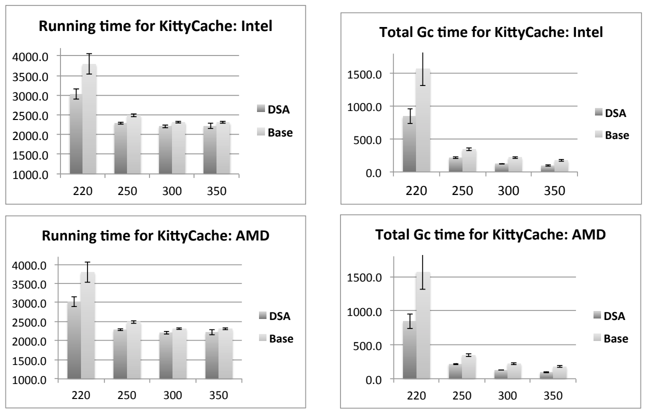


Figure 3. Total running time and total GC time (in milliseconds) for *KittyCache* stress test with 250K entries. The *x*-axis is the heap size used.

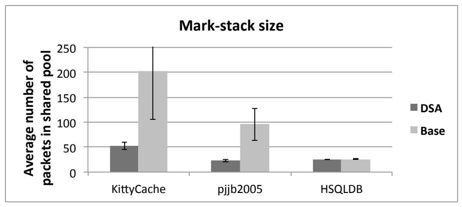


Figure 4. Number of work-packets in shared mark-stack.

imum shared pool size in each collection cycle. We then computed the average and standard deviation of the obtained record in the 3 executions and the results are depicted in Figure 4. For the pjbb2005 and kittyCache the mark-stack size was reduced by a factor of 3.85-4.15. For the HSQLDB benchmark the difference is very small, but the mark-stack size is rather small even in the baseline algorithm. The error bars represents standard variant. For the baseline algorithm the variance was very high, meaning that different collection cycles (in the same execution) requires different mark-stack size. In contrast, the variance is much smaller for the DSA memory manager.

Finally, we experimented with the DaCapo benchmark suite 2009. We were able to execute only 6 benchmarks: avrora, luindex, lusearch, sunflow, xalan and pmd. The other benchmarks could not be executed on the JikesRVM Java virtual machine. In avrora, lusearch, and xalan, a data structure used by the JikesRVM Java virtual machine was the dominant data structure. Therefore, we added the DSA interface to the JikesRVM data structure. In the pmd benchmark, a custom data structure is significant. However, the code semantics does not expose a data-structure-like interface, so adding the DSA interface to the data structure requires deeper understanding of the benchmark (if at all possible). Luindex makes an insignificant amount of allocation, and the GC load was low. In sunflow, a dominant data structure is an array of image samples, which is already GC friendly. Therefore, we did not add the DSA interface to any of the benchmarks in the DaCapo suite 2009, and modified only the JikesRVM internal data structure. For each of these benchmarks, we compared the running times before and after modifying the JikesRVM major data structure. We do the comparison for various heap sizes, namely 1.2x, 1.5x, and 2x of the minimal running heap size. The comparison is depicted in figure 5. For lack of space, the comparison for the AMD machine is not depicted. The average improvement in GC time is 11-16% and the average implemented in overall time is 1-2%.

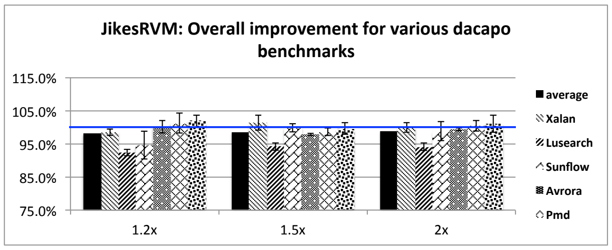
Note that the pmd benchmark suffers a slowdown due to our modification. The pmd data structure has a linked-list like structure, thus tracing it is poorly parallelized. A possible explanation to the slowdown is that in the original execution, a single thread traces this data structure while other threads trace other nodes, including the JikesRVM major data structure. In the modified execution, all threads start by scanning the JikesRVM data structure. Only later, a single thread traces the pmd data structure, which explains the slowdown in this case. Indeed, there was no slowdown where the basic memory manager was the mark-sweep and the mark-bits were put in a side table. We stress that we were not able to apply the DSA mechanism to the pmd program, but only to the JikesRVM that executes this program.

9. Related Work

The interactions between applications and garbage collection systems have not been explored much in the literature. Both Java and C# contain an interface for the program to invoke a garbage collection cycle. The use of these functions is usually discouraged. We are unaware of any work that discusses these functions.

An immediate free instruction that can be inserted by the compiler was studied by Cherem and Rugina [9], and Guyer et al. [15]. They suggested a compiler analysis technique that identifies unreachable objects. The compiler inserts an immediate free instruction for these objects, which reduces the number of garbage collection invocations, and improves efficiency. However, the compiler analysis is conservative. In contrast, an analysis of data structures as in this paper is difficult to perform automatically. Triggering garbage collection cycles when the live space is low is beneficial as there are less objects to trace. Buytaert et al. [8] proposed a triggering scheme with a pre-profiling stage that identifies locations of code where it is favorable to initiate garbage collection.

Aftandilian and Guyer [1] proposed a GC-application interface for evaluating user assertions. This interface allows the user to express assertions about heap layout, which the garbage collection evaluates at runtime. Wick and Flatt [29] proposed another GC-application interface for bounding the memory usage of a child processes. This allows the program to launch untrusted child processes without allocating a different heap for each child and without complicating the data transfer between the processes.



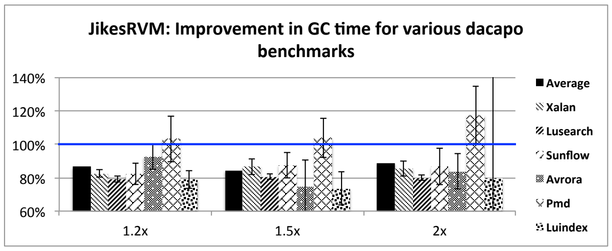


Figure 5. Comparing total running time and gc time for various benchmarks in the DaCapo suite. The figure only present the ratio between the modified JikesRVM and the unmodified Immix. Only a JikesRVM internal data structure was changed.

Recently, Reames and Necula [22] used "freeing hints". This method relies on deletions of all (or almost all) objects being specified by the programmer in the program.

Allocating objects based on their type was suggested by Shuf et al. [25]. They considered a memory manager that places frequently used objects in the young generation. This improved garbage collection locality, eliminated barriers, and improved space efficiency.

Deep heap shapes inherently limit parallelism opportunities. Barabash and Petrank [3] found that deep heap shapes do exist in practice. They also proposed two ideas for overcoming such heap shapes. Subsequently, Eran and Petrank [13] discovered that deep heap shapes are mostly caused by stacks or queues. They proposed a lock free queue implementation with low depth via shortcuts. Their solution does not apply to other linked list structures. The DSA collector eliminates the problem of deep heap shapes for all objects in the annotated data structures.

Avoiding mark-stack overflow was considered by Ossia et al. [21]. They divided the mark-stack into work packets, where every work packet contains a fixed number of objects. This allows the mark-stack to dynamically change its size, and to be dynamically distributed between different threads. It does not eliminate mark-stack overflow in the cases when there is not enough memory for additional work packets. Ugawa et al. [28] attempted to address mark-stack overflow by improving recovery time. Upon an overflow, they record the set of places where a visited untraced node can reside and use it for faster recovery from the overflow.

10. Conclusion

We proposed an interface between a program and the memory manager that allows special treatment for data structures. As data structures are often used to hold much of the program data, a data-structure aware (DSA) garbage collector can use knowledge about data structures in the program to improve performance. Both the program and garbage collector benefit from improved locality, and additionally the garbage collector benefits from improved scalability and a lower use of the mark-stack. We have demonstrated the ease of use for our interface and its effectiveness by using it with several data structures from the standard Java *java.util* package, as well as with the pSPECjbb2005 benchmark, the KittyCache program, and the JikesRVM Java virtual machine. The use of the interface only required a handful of modifications in the code and the performance improvements for the garbage collection and the program were dramatic. The KittyCache benchmark overall running time was improved by 4-20%; the pSPECjbb2005 benchmark overall running time was improved by 2.8-6%; the HSQLDB benchmark overall running time was improved by $31 - 32\%$; the DaCapo benchmarks, running over the JikesRVM that uses the DSA interface, were improved by 1-2% on average.

References

[1] E. E. Aftandilian and S. Z. Guyer. Gc assertions: using the garbage collector to check heap properties. In *PLDI*, pages 235–244, 2009.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] K. Barabash and E. Petrank. Tracing garbage collection on highly parallel platforms. In *ISMM*, pages 1–10. ACM, 2010.

[4] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27(6):1097–1146, 2005.

[5] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *OOPSLA*, volume 38, pages 344–358. ACM, 2003.

[6] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI*, volume 43, pages 22–32. ACM, 2008.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[8] D. Buytaert, K. Venstermans, L. Eeckhout, and K. De Bosschere. Gch: Hints for triggering garbage collections. In *Transactions on High-Performance Embedded Architectures and Compilers I*, pages 74–94. Springer, 2007.

[9] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. In *ISMM*, pages 138–149. ACM, 2006.

[10] J. E. Cook, A. L. Wolf, and B. G. Zorn. A highly effective partition selection policy for object database garbage collection. *Transactions on Knowledge and Data Engineering*, 10 (1):153–172, 1998.

[11] D. Detlefs and T. Printezis. A generational mostly-concurrent garbage collector. Technical report, Sun Microsystems, 2000.

[12] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for java. In *PLDI*, volume 35, pages 274–284. ACM, 2000.

[13] H. Eran and E. Petrank. A study of data structures with a deep heap shape. In *MSPC*. ACM, 2013.

[14] R. Garner, S. M. Blackburn, and D. Frampton. Effective prefetch for mark-sweep garbage collection. In *ISMM*, pages 43–54. ACM, 2007.

[15] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: a static analysis for automatic individual object reclamation. *PLDI*, pages 364–375, 2006.

[16] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *PLDI*, volume 40, pages 143–153. ACM, 2005.

[17] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. In *ISMM*, pages 73–84. ACM, 2004.

[18] H. Kermany and E. Petrank. The compressor: concurrent, incremental, and parallel compaction. In *PLDI*, pages 354–363, 2006.

[19] KittyCache. Kittycache. https://code.google.com/p/kitty-cache/, 2009.

[20] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for java. In *OOPSLA*, pages 367–380, 1999.

[21] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko. A parallel, incremental and concurrent gc for servers. In *PLDI*, pages 129–140, 2002.

[22] P. Reames and G. Necula. Towards hinted collection: annotations for decreasing garbage collector pause times. In *ISMM*, pages 3–14. ACM, 2013.

[23] N. Sachindran, J. E. B. Moss, and E. D. Berger. mc^2: high-performance garbage collection for memory-constrained environments. *OOPSLA*, 39(10):81–98, 2004.

[24] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley. Taking off the gloves with reference counting immix. In *OOPSLA*, pages 93–110, 2013.

[25] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–306, 2002.

[26] F. Siebert. Limits of parallel marking collection. In *ISMM*, pages 21–29. ACM, 2008.

[27] SPEC. Specjbb2005. http://www.spec.org/jbb2005/, 2005.

[28] T. Ugawa, H. Iwasaki, and T. Yuasa. Improvements of recovery from marking stack overflow in mark sweep garbage collection. *IPSJ Online Transactions*, 5, 2012.

[29] A. Wick and M. Flatt. Memory accounting without partitions. In *ISMM*, pages 120–130. ACM, 2004.

[30] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: the impact of zeroing. In *OOPSLA*, volume 46, pages 307–324. ACM, 2011.

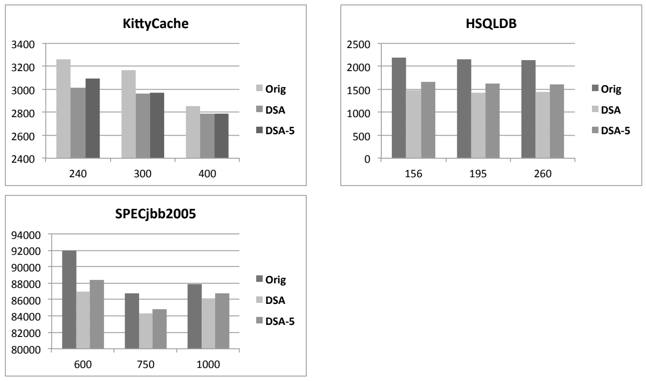


Figure 6. Throughput comparison for the original implementation, the DSA standard implementation, and an implementation that runs IdentifyLeaks once every 5 collection cycles.

A. Memory Leaks

When using the DSA interface, there is a possibility that the programmer will miss code locations in which nodes are removed from the data structure, potentially causing memory leaks. In this appendix we discuss our experience in this matter.

In the benchmarks we modified we used two methods to make sure that there were no memory leaks in the observed executions. First, we modified the IdentifyLeaks interface to check whether there existed an unreachable object that was not declared as removed. Second, we measured the number of occupied lines (which measures the amount of live memory) and ensured that it was similar in both the DSA and baseline execution.

The data structures that we modified are widely used in practice and their implementation consists of hundreds to thousands of lines of code. We did not study (or even read) the entire implementation. Instead, we searched for an internal class called "Node" or "Entry" and a function called "remove" or "delete". For all applications, except for the ConcurrentQueue library data structure, we successfully applied the DSA interface without any memory leaks in the first attempt. Namely, we never missed a deletion of objects from the data structure.

The one application for which we erred, was the ConcurrentQueue for which our first DSA version leaked memory. After inspecting the code of the remove function, which is actually a pop for the queue data structure, we discovered that when a node A is popped from the queue, it is not actually unlinked from the data structure. Instead, it becomes the new sentinel and the previous sentinel is the node that actually gets unlinked from the data structure. Incorrectly, we marked the currently popped node A as removed. The result of this mistake is that the first sentinel is never removed, and subsequently, all nodes in the queue becomes forever reachable from an un-removed data structure node. This meant that no object could be reclaimed. In our implementation this implied some frequent GC cycles (because not enough memory was freed) culminating in a cycle that was not able to proceed and then IdentifyLeaks was called. At that point, that first sentinel node was reclaimed and all other leaked nodes were reclaimed with it. From there on, performance was back to normal. Probably, the programmer of the queue would not make such a mistake.

On top of mistakes, there could also be applications and data structures for which it is easy to identify most of the removals but not all of them. We don't expect this to be frequent, but for such cases there is an easy solution. It is possible to call IdentifyLeaks once every k collection cycles. In Figure 6 we measure the performance overhead with a typical parameter $k = 5$. Orig stands for the baseline implementation with no DSA, DSA stands for the DSA benchmark when IdentifyLeaks is called only when necessary, i.e., when an out-of-memory exception is about to be thrown, and DSA-5 stands for the DSA benchmark when IdentifyLeaks is called once every 5 collection cycles. As in 20% of the collections the DSA interface is not used, the advantage is reduced by 19-29% depending on application and heap size.