The Single-Referent Collector: Optimizing Compaction for the Common Case

MICHAL WEGIEL and CHANDRA KRINTZ
University of California, Santa Barbara

15

Compactors that move or copy objects need to adjust pointers. In extant compactors, pointer adjustment involves inspecting every pointer in the heap and computing the target address for each pointer. At the same time, in modern Managed Runtime Environments (MREs), only a fraction of pointers in the heap changes during compaction. This is because state-of-the-art MREs do not compact the prefix of the heap that contains few dead objects, allowing gaps between live objects and tolerating small space overhead.

We describe the design and implementation of the Single-Referent Collector (SRC), a new compactor that reduces the cost of pointer manipulation by avoiding inspection and adjustment of pointers that do not change. SRC exploits the fact that in modern applications, most live objects have only one incoming pointer. For such objects, SRC stores the address of the referent in the object header. Only objects that move have their referent inspected and adjusted. The remaining pointers in the heap are not processed. SRC uses an overflow table to handle objects with multiple incoming pointers.

We investigate a number of standard benchmarks and open-source applications to substantiate key statistical observations that underlie the design of SRC. We implement SRC in the HotSpot JVM as part of a generational collection system and compare it empirically with the Lisp2 compactor. We find that, by decreasing the cost of pointer processing, SRC enables significant reduction in pause times and improves application throughput.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Memory Management (Garbage Collection)*

General Terms: Design, Performance, Experimentation, Algorithms

Additional Key Words and Phrases: Compaction, garbage collection, virtual machines

ACM Reference Format:

Wegiel, M. and Krintz, C. 2009. The single-referent collector: Optimizing compaction for the common case. ACM Trans. Architec. Code Optim. 6, 4, Article 15 (October 2009), 26 pages. DOI = 10.1145/1596510.1596513 http://doi.acm.org/10.1145/1596510.1596513

1. INTRODUCTION

Managed Runtime Environments (MREs) for object-oriented, type-safe programming languages, such as Java or C#, implement automatic memory

Authors' address: mwegiel@cs.ucsb.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1544-3566/2009/10-ART15 $10.00
DOI 10.1145/1596510.1596513 http://doi.acm.org/10.1145/1596510.1596513

management (garbage collection, GC) to provide memory safety and simplify software development. Among extant GC algorithms, compacting collectors [Jones 1996] are one of the most commonly used in existing production MREs. This is because compactors eliminate heap fragmentation, improve mutator locality, and enable simple and efficient linear (bump-pointer) object allocation [Wilson 1992].

The key limitation of extant compactors is suboptimal pointer adjustment. Most state-of-the-art compactors move some objects in the heap and, therefore, need to adjust pointers. Reducing the amount of copying/moving has been the subject of many studies. As a result, many collectors move only part of a heap, for example, the Pauseless GC [Click et al. 2005] and Immix [Blackburn and McKinley 2008] perform block-based partial evacuation, while Lisp2 [Jones 1996] and the HotSpot GC [HotSpot JVM GC] use a dense prefix to limit sliding compaction only to the region with many dead objects. However, existing GCs still process all the pointers in the heap, despite the fact that typically only few pointers change (as many objects do not move). This processing can be expensive [Wegiel and Krintz 2008] because of numerous memory accesses (and its impact on memory hierarchy performance) and the computation [Kermany and Petrank 2006] that accompanies determining the new value of each pointer.

Herein, we investigate how to reduce the cost of pointer adjustment to nearly optimal. We design a new compactor that iterates only over the pointers that need to be updated. To date, this problem has received relatively little attention.

Extant collectors perform pointer adjustment either through forwarding pointers (e.g., Lisp2) or with the help of a marking bitmap and an offset table (e.g., the Compressor [Kermany and Petrank 2006] and the HostSpot GC [HotSpot JVM GC]). In the former approach, a new value of a pointer is retrieved from the header of the target object, which contains a forwarding pointer. The latter technique is more complex, as it computes the new pointer value by traversing a small fragment of a marking bitmap (bounded by a block size) and summing up the total size of live objects in a given block and in the preceding blocks (using an offset table). Forwarding pointers are simpler and faster but require an additional pass over the heap (to install forwarding pointers). A marking bitmap eliminates the need for an additional pass, but for each pointer update, a time-bounded but complex and relatively expensive processing needs to be done. Clearly, reducing the number of pointer updates is more beneficial for collectors using a marking bitmap. However, our evaluation shows that even when forwarding pointers are used (e.g., in Lisp2), the performance gain due to the avoidance of processing of all pointers can also be significant (over 10% execution time).

The key idea in the single-referent collector (SRC) that we contribute herein is to reverse the direction of pointers in the object graph, similarly to the way Jonkers performs pointer threading [Fisher 1974; Morris 1978; Jonkers 1979]. Instead of iterating over all pointers in the heap and looking for pointers that need to be updated, we first identify objects that move and then adjust their referents. SRC can be parallel and used in a generational GC [Lieberman and Hewitt 1981] system. The ideas underlying SRC can be applied to other GC

algorithms to optimize pointer adjustment. The design of SRC is based on three statistical observations.

—Most live objects in modern applications have only one referent [Deutsch and Bobrow 1976; Dieckmann and Hölzle 1998; Blackburn et al. 2006].

—A large part of the heap is not moved during compaction in state-of-the-art MREs. This is because MREs aim at avoiding low-yield compactions that do not free enough space to justify the cost of object moving [Wegiel and Krintz 2009]. Thus, at the cost of small space overhead, controlled by a command-line parameter, they compute a dense prefix, a heap region that contains mostly live objects, and do not compact it. A special treatment of dense blocks is also reported in Kermany and Petrank [2006].

—Most MREs reserve a machine word for an object header, which contains such information as locking state and hash code (among others). The vast majority of object headers remain unused during program execution (as most objects are unlocked at any given point in time). Thus, most headers are available for reuse by GC (for forwarding pointers or, like in SRC, for pointer reversal).

During marking, SRC reuses object headers to reverse as many pointers as possible in place (i.e., without any space overhead). For all live objects that have exactly one referent, SRC stores the address of the referent in the object header. Objects with more than one referent, use an overflow table—one referent is stored in the object header and the addresses of the remaining referents are remembered in the overflow table for later processing. Once marking is done, SRC determines the target location for each object. As soon as the target location is known, the referent is updated if the object in question changes its address. Since most objects do not move, most pointers are not updated. SRC imposes a small space overhead, due to the overflow table, but its overhead can be bounded by a fallback to Lisp2 (determined dynamically).

In summary, we make the following contributions:

—Comprehensive empirical analysis of modern benchmarks and compaction strategies that provides evidence for the key statistical observations that we build upon when designing SRC.

—Discussion of how extant compactors can optimize for the common object graph shape. State-of-the-art compactors do not exploit the single-referent property. Extant GC systems strive to reduce the cost of compaction by moving only part of the heap; however, all pointers still need to be processed.

—Design of SRC, a compactor that reduces the cost of pointer processing/manipulation to nearly optimal. Unlike extant compactors, SRC almost never processes (where processing includes reading the current and computing the new value of) pointers that do not change. To date, optimization of pointer manipulation has received little attention.

—Implementation of SRC in the HotSpot JVM in a generational collection framework. Empirical evaluation of SRC based on standard benchmarks and

open-source applications demonstrating that SRC improves performance and responsiveness compared to the Lisp2 algorithm while being simpler.

In the sections that follow, we discuss the background (Section 2), describe the design and implementation of SRC (Sections 3 and 4), and present the results of its experimental evaluation (Section 5).

2. BACKGROUND

In the following text, we provide a detailed background on two compactors: Lisp2, as we compare SRC with it, and Jonkers, as SRC reverses the object graph, a technique also employed by Jonkers. In addition, we discuss other state-of-the-art compactors in the context of how they perform pointer adjustment, which is the focal point of SRC.

In our description of the algorithms, we assume that objects are ordered in the heap from left to right. Therefore, if object o precedes object p in a sequential scan over the heap, object o is to the left of p. A backward pointer originating from object o points to an object to the left of o. Similarly, a forward pointer located in object p points to an object to the right of p. Note also that a forwarding pointer is a pointer stored in an object header and pointing to a location to which the object moves. We often use the terms pointer threading and pointer reversal interchangeably—they both refer to the process of associating a list of incoming pointers with an object.

An overview of several classic compaction algorithms for uniprocessors can be found in Cohen and Nicolau [1983]. Table-based compactors operate in three phases but have the worst-case complexity of $O(n\log n)$. This is because pointer readjustment is done based on a binary search in a table that stores sorted offsets to the clusters of live objects. Among linear-time serial compactors, the most important ones are Lisp2 and the Jonkers algorithm. The serial Lisp2 collector is suitable for client-side desktop machines (e.g., the HotSpot JVM uses it for full-heap compacting garbage collection in the client mode). Lisp2 has a parallel version that targets multiprocessors [Flood et al. 2001]. Both Lisp2 and the Jonkers algorithm are serial stop-the-world sliding mark-compact collectors that preserve object order and compact all live objects into a single contiguous area in the (lower part of the) heap.

2.1 Lisp2

Lisp2 is a four-phase compactor. The goal of the first phase (called marking) is to find all live objects by following the object graph starting from the roots. The second phase is a sequential scan over the heap that installs forwarding pointers in the headers of live objects. Each forwarding pointer determines the target location for a particular object after compaction. A target location for an object x is computed as the total size of live objects that precede x. The third phase amounts to a sequential traversal over the heap and pointer adjustment. Each pointer is set to the value stored in the header of the object that it currently points to. The fourth phase involves moving subsequent live objects to the locations specified by their forwarding pointers.

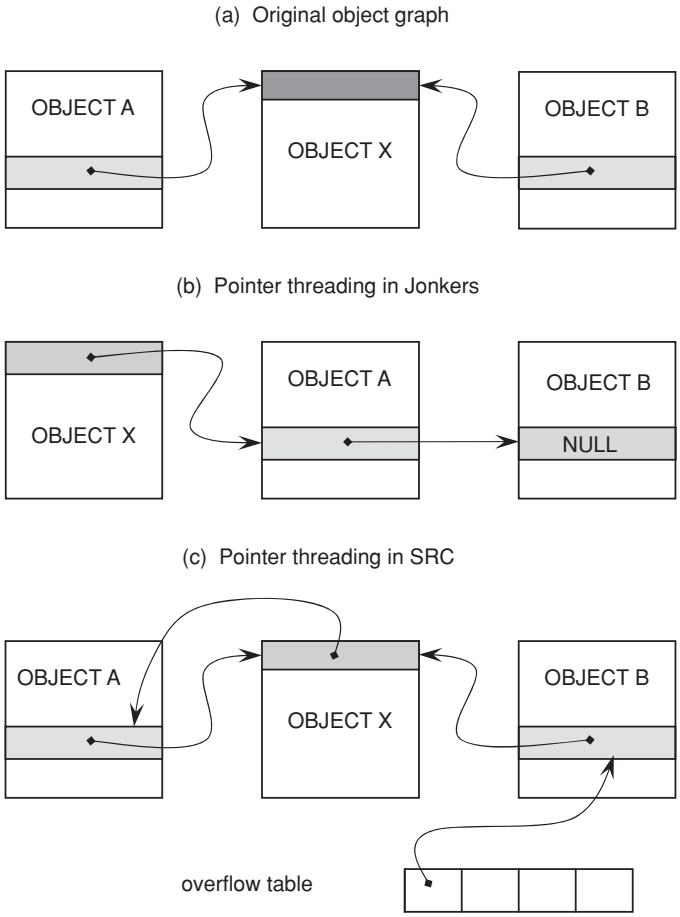

Fig. 1. Comparison of pointer threading (reversal) in the Jonkers GC (b) and in SRC (c) for an example object graph (a). Note that SRC does not overwrite any pointers, while the Jonkers algorithm overwrites all pointers. Outgoing object pointers are shaded light gray and object headers are shaded dark gray.

2.2 Jonkers

The Jonkers algorithm has three phases. Similar to Lisp2, the first phase is devoted to computing the transitive closure of the root set. The second phase is a sequential scan over the heap that performs pointer threading and partial-pointer adjustment. Pointer threading is a technique that chains together pointers that reference a single object. Figures 1(a) and 1(b) shows an example object graph before and after pointer threading. In the Jonkers collector, pointer threading is done in-place, meaning that it does not require any additional memory. In the second phase, the GC threads all pointers as it encounters them during the sequential scan of the heap. In addition, the system adjusts incoming forward pointers that have been threaded so far. When an object x is reached, all forward pointers that reference x have already been threaded by the algorithm and can, therefore, be updated (and unthreaded). By the end

of the second phase, the system has threaded all backward pointers (forward pointers were threaded too, but they are already updated and unthreaded). The third phase updates only incoming backward pointers. When the pointers are updated, the algorithm moves the currently processed object (at which point it is guaranteed that its fields are not part of any chain). Since object headers are occupied by pointers to chains, the algorithm must compute the target location for each object in both Phases 2 and 3 (as forwarding pointers are not used). The Jonkers collector requires only three phases but proves expensive in practice due to its extensive pointer manipulation.

2.3 Pointer Processing

All extant compactors process all pointers in the heap, regardless of whether a particular pointer changes. State-of-the-art compactors that move objects essentially use two techniques for pointer adjustment, one based on forwarding pointers and the other based on a marking bitmap.

Compactors employing forwarding pointers, such as Lisp2 and SRC, use the object header to store the address to which a particular object moves. Pointer adjustment in such a setting is simple and efficient—it amounts to reading the value of a forwarding pointer and updating the referent. The disadvantage is that forwarding pointer installation requires a separate pass over the heap.

Recent compactors, such as the Compressor and the HotSpot compactor, use a marking bitmap for pointer adjustment. The marking bitmap reserves 1 bit per heap word. During object graph tracing, GC sets bits in the bitmap to indicate locations of live objects. The heap is divided into blocks. GC computes per-block liveness statistics (i.e., how many bytes are live in a given block). Object moving and pointer updating are done simultaneously. For each pointer, its new value is computed using the per-block statistics and traversing a small fragment of the marking bitmap—the GC sums up the total size of live data before a specific block and within that block up to some object. Thus, in bitmap-based compactors pointer adjustment is relatively expensive; however, this cost is amortized by the avoidance of one pass over the heap.

The Pauseless GC uses a hash table (side arrays) for storing forwarding pointers for the currently relocated virtual pages. A hash table look-up is necessary to compute the new value of a pointer. Such a look-up executes as part of a read barrier.

3. COMPACTOR DESIGN

SRC exploits the widely known statistical property that in modern programs, most live objects in the heap have exactly one referent. Our empirical analysis of Java benchmarks and applications, as described in detail in Section 5, shows that across 25 programs, 91% of live objects in the heap have only one incoming pointer.

The key idea behind the SRC algorithm is that it reverses the direction of pointers during marking and subsequently uses the reversed object graph to update the pointers to the objects that change their location. Because of the single-referent property, pointer reversal can be done nearly in-place (almost

without space overhead). SRC reuses the object header to store the address of one (and in most cases the only) referent for each object. State-of-the-art MREs employ object headers to implement object locking, store object hash codes, GC mark bits, and forwarding pointers. At any given moment, most objects have their header in a canonical state (i.e., unlocked and unforwarded); and therefore, very few headers need to be preserved when GC occurs.

Pointer reversal takes place during marking so that SRC can exploit reference locality and avoid introducing memory accesses that might result in a cache miss. An additional advantage of piggybacking on marking is that there is no need for a dedicated pass over the heap. Thus, pointer reversal can be implemented with small overhead.

SRC reverses the object graph to avoid processing for pointers that are not adjusted. Once a new location of an object is known, the referent can be properly updated. There is no need to traverse all pointers to identify those that need adjustment.

SRC employs an overflow table to handle popular objects (i.e., those with more than one referent). In the object header, the address of one referent is stored. The remaining referents are appended to the overflow table during marking. Since there are few popular objects, the imposed space overhead is small in practice (we evaluate it in detail).

A key observation underlying SRC is that extant MREs rarely compact the whole heap. For example, the HotSpot JVM finds a dense prefix, a region at the beginning of the heap that contains few dead objects, and excludes it from compaction. The rationale is avoidance of low-yield and expensive compactions. In the dense prefix, dead objects are not reclaimed, and the resulting gaps incur space overhead. This overhead is controlled by an MRE parameter, which specifies what percentage of the heap can be designated for the unused dead space. Our empirical evaluation of Lisp2 in the HotSpot JVM shows that across the benchmarks that we use, and for the allowed dead space of 5%, 72% of objects do not move.

The goal of SRC is to reduce the cost of pointer adjustment to nearly optimal so that the pointers referring to nonmoving objects are not processed at all. Existing compactors always process all pointers (i.e., read each pointer and compute its new value) despite the fact that most objects do not move. Pointer processing can be expensive, even when pointer forwarding is used, not to mention marking bitmaps.

SRC comprises three phases: marking/threading, object forwarding/pointer adjustment, and object moving. Lisp2 has four phases because it needs a dedicated phase to process all the pointers in the heap and update those that change. SRC is simpler and more efficient.

3.1 Phase I: Pointer Threading

SRC performs pointer threading during the marking phase (unlike the Jonkers algorithm) for simplicity and locality of reference. SRC threads all pointers, regardless of whether the incoming reference belongs to an object in the heap (i.e., is a reference field) or lies outside of the heap (i.e., is a root). The only

Table I. Marking and Pointer Threading (Phase I)

```
 1: save headers with locking/hash code
 2: pending ← root set
 3: while pending ≠ ∅ do
 4:   r ← pending.extract()
 5:   if object(r) is not marked then
 6:     mark(object(r))
 7:     object(r).header ← r
 8:     for p in references(object(r)) do
 9:       pending.insert(p)
10:     end for
11:   else
12:     e ← new entry (overflow table)
13:     e.reference ← r
14:   end if
15: end while
```

exception to this rule is the class pointer that points to an object that encapsulates the metadata that defines the object type. This exception applies only to JVMs that treat objects and metadata in a uniform way (i.e., classes are represented as objects in the heap, which leads to several levels of metadata hierarchy ending with a circular dependency). The rationale for excluding class pointers from threading is that a particular metaobject is always pointed to by all objects of the type that it represents. Metaobjects, therefore, are referred to by a large number of other objects, and as such, are not good candidates for pointer threading due to the space overhead they would cause in the overflow table.

While tracing the object graph (typically using depth-first search) and marking the reachable objects, SRC reverses pointers. Object headers are used to store the addresses of referents. In case of multiple referents, SRC employs the dynamically expanded overflow table. An example object graph and its threaded representation are shown in Figure 1(a) and (c).

Table I lists the pseudocode for Phase I. The set of addresses of pointers to be processed is kept in pending, which in practice is a stack or a queue. We extract subsequent addresses of pointers (r in the procedure code) from the pending set (by calling `extract`) and dereference each pointer address twice to obtain an object address (our notation for this operation is object(r)). Objects visited for the first time are marked, and their headers are used to store the address of the referent (r). In case of already-marked objects, the address of the referent is appended to the overflow table. Given the single-reference property that is common to Java programs, we expect to require very few entries in the overflow table. Accessing this table imposes both space and time overhead (it degrades locality during tracing).

3.2 Phase II: Pointer Forwarding

At the start of the second phase, in the header of all live objects is an address of one of their referents. Pointers not recorded in the object headers are in the

Table II. Object Forwarding and Pointer Adjustment (Phase II)

```
 1: o ← first live object
 2: while o ≠ nil do
 3:    h ← o.header
 4:    f ← forward(o)
 5:    o.header ← f
 6:    if o is relocated then
 7:       referent(h) ← f
 8:    end if
 9:    o ← next live object
10: end while
11: for e in entries(overflow table) do
12:    p ← e.reference
13:    f ← p.header
14:    address(p) ← f
15: end for
```

overflow table. As detailed in Table II, we iterate over subsequent objects (o is the address of the current object) and compute a forwarding pointer for each live object (we call the `forward` function to accomplish the latter). To compute the next live object, we sequentially skip over dead objects until we find the next marked object. The `forward` function computes the location an object will occupy after compaction. If a particular object moves, we overwrite the pointer whose address has been stored in the object header (h) with the object's destination address (f). Once the referent (h) is updated, we use the object header to store the value of the object's forwarding pointer (f). SRC needs forwarding pointers to properly update pointers that overflowed (i.e., those from the overflow table). The last step in Phase II is iterating over the overflow table and adjusting the subsequent pointers. At this point, all object headers contain forwarding pointers.

SRC handles class pointers specially—they are not threaded and SRC updates them using forwarding pointers. In consequence, Phase II must be first completed for all metaobjects (so that forwarding pointers are installed) before we start Phase II for regular objects. SRC updates each class pointer by reading the forwarding pointer from the header of the instance of the class. This is a method similar to the one used in the Lisp2 algorithm for all pointers.

Unlike Lisp2, SRC need not identify and traverse individual reference fields of each object in Phase II. The information that SRC gathers during marking is sufficient to adjust the pointers. This obviates the need for accessing metadata for each object (in Phase II) to determine which fields have reference types (what makes Phase II very efficient).

3.3 Phase III: Object Moving

After SRC identifies the target locations for the objects that must be moved and adjusts the relevant pointers, SRC moves the objects to the left end of the heap (Table III shows the pseudocode). SRC copies each object (o) to the location specified by the forwarding pointer that is stored in the object

Table III. Object Moving (Phase III)

```
1: o ← first live object
2: while o ≠ nil fo
3:    if o needs to be relocated then
4:       move and clear header(o, o.header)
5:    else
6:       clear header(o)
7:    end if
8:    o ← next live object
9: end while
10: restore saved headers
```

header (o.header). Objects in the dense prefix do not move. All objects in the heap have their headers cleared (forwarding pointers are removed). Once moving completes, the headers that contained locking state or hash codes are restored. A separate table is used to store such headers. Typically, few headers need saving before GC, so this does not impose large time/space overhead.

3.4 Space-Bounded SRC

SRC can fall back to Lisp2 to guarantee a specific space overhead bound. The overflow table is filled during marking; therefore, fallback is possible starting from Phase II. When SRC reaches the limit on the number of entries in the overflow table, threading is suspended, marking proceeds until it is finished, and the collector continues its operation as Lisp2. The implementation is straightforward as the output of marking expected for Lisp2 is the subset of what SRC generates in Phase I.

3.5 Generational SRC

SRC can be implemented in MREs with heaps consisting of multiple generations. For example, our implementation uses three generations: young, old, and permanent. In addition, the young generation consists of three memory regions: eden, from space, and to space.

As described previously, SRC's correctness requires metadata to be processed first in Phase II. In configurations with a dedicated generation for metadata (which is typically called a permanent generation), Phase III also must be completed in the metageneration before the compactor can move on to the remaining generations. This is because once class pointers are updated in Phase II, they point to incorrect locations until metaobjects are actually moved in Phase III. Regular generations can be processed in any order as long as the metageneration is traversed first in both Phases II and III.

Note that metaobjects also have class pointers (which point to metametaobjects). However, class pointers in the metageneration always point to the left. The reason for this is that whenever we allocate a metaobject the corresponding metametaobject must have been already allocated. Assuming left-to-right allocation, all class pointers in the metageneration point to the left. In consequence, a sequential traversal (accompanied by moving) from the left to the

right end in Phase III guarantees that the class pointer of the currently processed metaobject is always correct.

An important advantage of scanning the metadata generation first in Phase II is that by the time we start traversing regular generations, we know whether any metaobject has been moved. If no objects have been relocated in the metageneration (which in our empirical configuration is always the case), then it is not necessary to forward class pointers in Phase II, which translates to considerable reduction of work, as every object has a class pointer.

SRC can be used with different heap configurations, consisting of multiple memory regions (either generations or moving and nonmoving spaces, such as large object spaces). Except for the previously described ordering constraints pertaining to the metageneration, the remaining spaces can be processed in any order, and there are no additional requirements on their number, size, and object moving strategy (i.e., whether a particular space is compacted).

3.6 Parallel SRC

We have extended SRC to support multiple GC threads in order to scale on multicore and multiprocessor systems. Parallel SRC comprises three phases.

The first phase, marking, can be parallelized in a way similar to Lisp2 [Flood et al. 2001]. Parallel GC threads maintain thread-local overflow tables to avoid synchronization and interthread contention. While computing the transitive closure of the objects reachable from the roots, GC threads strive to load balance the available work. We employ dynamic work stealing for this purpose. A GC thread acquires an object reference either from its local queue, or by retrieving an entry from a queue that belongs to another thread. Marking is terminated after all GC threads declare that they are idle. Object marking and pointer threading require an atomic compare-and-swap operation on an unmarked object header when the object is visited by multiple racing GC threads at the same time. The GC thread that wins the race marks the object and stores the referent address in the header. Since parallel SRC uses per-thread overflow tables, pointer reversal does not introduce any additional synchronization to marking. We do not use a marking bitmap.

Similarly to the parallel HotSpot compactor, we divide the heap into fixed-size blocks. Marking computes basic per-block statistics, that is, how many bytes are live in the block and an offset of the first live object in the block. In the second phase, GC threads divide the heap into several chunks, as many as GC threads. Each chunk comprises multiple blocks and covers similar amount of live data. Each GC thread forwards objects and updates referents stored in the headers in its own chunk. After all GC threads finish object forwarding, they execute a barrier (join point), and each GC thread proceeds to adjust pointers in its own per-thread overflow table.

In the third phase, GC threads claim available blocks atomically and fill them with live objects in parallel. A block becomes available when all its objects have been evacuated (it is empty, i.e., its live data counter is zero) or it has been compacted onto itself. Filling a block involves identifying the source block(s) and copying the objects destined for the block until the block is full or no more

live objects are left. Per-block statistics are updated as blocks are filled (live data counter is decreased, the first live object offset advances to the right). Initially, the first block in the heap gets compacted. Then, it becomes available and is a filling target for potentially many other blocks. Compaction starts off single-threaded and gradually increases its degree of parallelism as multiple blocks become available for claiming by GC threads.

4. IMPLEMENTATION

We have implemented SRC in HotSpot [OpenJDK], an open-source (GPL), production-quality, high-performance Java Virtual Machine from Sun Microsystems written in C/C++ (source code version 7-ea-b10, released on 3/21/2007). SRC is a serial mark-compact collector that is invoked during major (full-heap) collections that involve compaction of all generations.

4.1 HotSpot Memory Management

HotSpot uses handleless objects, meaning that all object references are direct pointers (there is no level of indirection). Every object has a two-machine-word header. SRC reuses this header and does not require any additional header space. The first word (called mark) contains object age bits, identity hash code, as well as synchronization/locking bits. The lowest two bits are set to 1 to indicate the fact the an object is marked. The mark word is also used to store a forwarding pointer during mark-compact GC. The second word (called class pointer) points to an instance that contains reflective metadata. HotSpot uniformly represents metadata as objects and maintains a self-referential three-level metadata hierarchy where the root object describes its own type. Arrays contain an additional (third) compulsory word to store the array length.

HotSpot uses a generational heap layout. There are three generations: permanent, tenured (old), and young. The young generation is further subdivided into eden and two equally sized survivor spaces (called from-space and to-space). Objects are initially allocated in the eden (if their size does not permit that, the allocation is done directly in the tenured generation). Within the young generation, a copying collector (called scavenger) is used. The scavenger evacuates live objects from eden-space and from-space to to-space and promotes objects that survived several minor collections to the tenured generation. To reduce synchronization costs, the system assigns each thread a thread-local allocation buffer (TLAB) from the eden space.

The HotSpot old generation supports several different garbage collectors. In the client mode, by default, it uses a standard four-phase Lisp2 mark-compact collector. The system performs stop-the-world collection by a dedicated thread upon memory exhaustion by the application and after reaching a safepoint at which all mutator threads are suspended. The marking phase involves traversing all reachable objects (across all generations) in a depth-first order and marking them (i.e., setting the least-significant two bits of the mark word to 1). The GC saves object headers that contain active locking state or a nonzero hash code prior to full collection and restores them afterward (to avoid overwriting

by a forwarding pointer). The GC preserves relevant headers in a dedicated dynamic array the length of which is extended on demand.

4.2 Pointer Threading and the Overflow Table

Prior to marking, SRC secures space for the reversed pointers to objects that have more than one incoming reference. The key advantage of the overflow table over the Jonkers approach is that it enables SRC to avoid many pointer manipulations. SRC employs a dynamic array for this purpose, following the design of existing auxiliary data structures (e.g., the marking stack). The overflow table does not require a contiguous memory region, its functionality is essentially a set of linked-list operations. SRC can grow the array one chunk at a time or exponentially (we use the latter option). SRC stores the overflow table outside of the heap. The table contains addresses of pointers that point to objects with more than one referent. To bound the space overhead introduced by this auxiliary data structure, SRC falls back to Lisp2 compaction. In our empirical evaluation, however, such fallbacks are never necessary by the benchmarks that we study, and space overhead is small.

In HotSpot, objects are aligned at word boundaries. Therefore, on 32-bit machines, the lowest 2 bits of each pointer are always cleared. HotSpot (like other JVMs) exploits this property to fit both marking bits and a forwarding pointer into the object header.

SRC makes use of the object alignment in a similar manner. After the marking phase, the headers of live objects point to locations of pointers in the heap, which are guaranteed to be word aligned. SRC overflows root pointers because in HotSpot roots (which include C heap pointers and operand stack entries) are not necessarily word aligned.

A potential improvement over this strategy would be to dynamically check at runtime whether a particular root is word aligned. However, we have not evaluated the impact of such an optimization experimentally.

SRC performs pointer threading in all generations. In the permanent generation, however, class pointers are not threaded since doing so would create significant space overhead. Class pointers are adjusted during Phase II via forwarding pointers.

5. EVALUATION

We compare SRC against the serial four-phase mark-compact Lisp2 collector that is employed by the HotSpot JVM by default in the client-compiler mode for full-heap (major) collections that compact all generations. For minor collections, HotSpot uses a serial copying collector in the young generation. SRC is implemented in all generations (i.e., the young, old, and permanent). Both Lisp2 and SRC take one parameter: dead space ratio, which determines the maximum allowed percentage of free space in the old generation (this parameter does not apply to the young and permanent generations). Higher values of the dead space ratio result in a larger dense prefix and a smaller number of moved objects.

We evaluate SRC's impact on execution time and full GC pause times, as well as investigate its space overhead. We analyze 25 benchmarks in the context

of the statistical properties that SRC exploits (nonmoving objects and single-referent objects). In addition to summarizing our experimental results across the heap sizes, we present more detailed empirical data for six selected benchmarks (three that perform well with SRC and three that only moderately benefit from SRC).

5.1 Methodology

Our experimental platform, is a dedicated dual-core Intel Core 2 Duo (Conroe B2) machine clocked at 2.66GHz with the unified 4M 16-way L2 cache and 32K 8-way L1 cache, 2GB main memory, running Debian GNU/Linux 3.0 configured with the 2.6.17 kernel. We use OpenJDK 1.6 and HotSpot version 7-ea-b10 compiled with GCC 3.2.3, in the optimized client-compiler (C1) mode.

For each experiment, we use a fixed size heap. The heap comprises three generations: the young, old, and permanent. When reporting heap size, we sum up the size of all three generations. We investigate five heap sizes for each benchmark/application ranging from the minimum heap size (and high GC activity) to the heap size with medium/minor GC frequency. We repeat each experiment seven times and report average values (arithmetic mean) along with standard deviation.

The young generation is 25% of the old generation. Survivor spaces (the `from-space` and `to-space`) occupy 33% of the young generation (the remainder is used by the `eden`). The permanent generation is 12MB (HotSpot default). We disable all explicit GC invocations and adaptive generation resizing.

During full collections, SRC compacts all three generations. In the young generation, no dead space is allowed (perfect compaction always takes place), while in the permanent generation, we allow any amount of dead space (objects are never moved). In the old generation, we vary the dead space ratio between 0% and 10% (via the SRC command-line parameter).

Our evaluation is based on 25 Java programs, which include standard Java benchmarks and open-source Java applications [GPLJava]. We use all DaCapo [Blackburn et al. 2006] and SPEC JVM'98 benchmarks. In addition, we employ SPEC PseudoJBB'00 [SPEC] and VolanoMark [VolanoMark] as well as five GPL applications. Individual benchmarks are described in Table IV where we also report performance data obtained using the Lisp2 collector for the 5% dead space ratio in the old generation (HotSpot default). In subsequent columns, we show heap size ranges, execution times, and GC cost for full-heap collections. We do not report any statistics for minor (young generation) collections as SRC is implemented for full-heap collections only and does not impact minor GC.

We run the default variants of the DaCapo benchmarks and use the input size of at least 100 for JVM'98. We execute VolanoMark with 42 chat rooms for 100 iterations and SPEC PseudoJBB with 5 warehouses for 10^5 iterations.

We evaluate SRC and Lisp2 using common metrics of GC performance: application throughput and maximum/average pause times. In addition, we investigate the number of incoming references for objects in the heap, as well as

Table IV. Baseline Statistics for the Java Benchmarks that we use Obtained Using the Lisp2 Compactor

Benchmark Program	Heap Size [MB]	Execution Time [s]	Full GC Time [s]	Full GC Count	Full GC Cost [%]	Program Description
DaCapo'06 Benchmarks						
antlr	14–18	2.2	0.2	13	7.0	parser generator
bloat	14–18	7.2	0.1	6	1.4	bytecode analyzer
chart	27–31	6.2	0.6	15	9.2	graph plotter
eclipse	34–42	26.7	1.1	16	4.0	IDE tester
fop	19–23	11.5	9.7	237	83.8	XSL parser
hsqldb	94–102	9.8	6.8	18	69.5	in-memory database
jython	14–18	6.4	0.1	6	1.6	Python interpreter
luindex	15–19	7.6	0.4	34	4.7	document indexer
lusearch	14–18	5.4	0.1	7	1.5	text search engine
pmd	29–33	6.1	1.7	34	27.4	source analyzer
xalan	31–39	6.7	1.8	58	27.4	XML converter
JVM'98 Benchmarks						
compress	33–37	2.7	0.1	13	3.0	LZW packer
db	23–27	8.1	1.0	33	12.6	in-memory database
jack	14–18	1.8	0.0	8	2.7	parser generator
javac	23–27	5.7	3.4	89	58.8	Java compiler
jess	14–18	1.2	0.0	2	1.5	expert shell system
mtrt	22–26	2.7	1.7	65	63.2	parallel raytracer
raytrace	14–18	3.7	3.0	194	81.5	3D renderer
GPL Applications						
beautyj	63–67	4.0	2.3	18	57.6	source transformer
findbugs	63–71	30.5	19.5	114	64.1	bug detector
jaranalyzer	14–18	4.6	0.1	8	1.6	JAR analyzer
javaguard	19–23	4.2	0.8	24	17.8	bytecode obfuscator
jdepend	31–35	13.1	1.7	34	13.1	dependency analyzer
Other Benchmarks						
psjbb	110–122	47.4	29.1	192	61.3	3-tier DB system
volano	29–33	41.1	8.6	138	21.0	chat server

We group benchmarks based on the suite/category they belong to. For each benchmark, we report the heap-sized range in Col. 2 (we use 5 regularly-spaced heap sizes across this range for each benchmark; heap size includes all generations: young, old, and permanent). Next, in Cols. 3–6, we present per-benchmark performance data for the minimum heap sizes: total execution time, total full GC time (minor collection are not included as SRC is implemented only for full-heap collections), the number of full GCs per run, and total full GC time relative to total execution time. The last column provides a brief description of each benchmark. We report arithmetic mean from 7 runs.

report the percentage of moved objects depending on the dead space ratio. For SRC, we also measure space overhead.

5.2 Object Graph

The efficacy of SRC, both in terms of its space overhead and pause time reduction, depends mostly on the single-referent property. In Table V, we report basic statistics for the object graph shape for each benchmark. All the measurements have been obtained for the old generation. Across the benchmarks, the percentage of objects with exactly one incoming pointer (second column) averages at 91%. Although the maximum (of the maxima) of referents to live objects (Column 4) is occasionally high, the expected number of incoming pointers (Column 3) is 1.1.

Table V. Per-Benchmark Statistics Related to the Live Object Graph

Benchmark Program	Single Referent [%]	Average Incoming	Maximum Incoming	Space Overhead [%]
antlr	90.0	1.3	772	3.0
bloat	91.2	1.3	788	4.2
chart	99.8	1.0	86	3.9
eclipse	91.5	1.3	2,614	6.3
fop	77.8	3.5	11	4.4
hsqldb	100.0	1.0	1	8.4
jython	95.1	1.2	3,040	5.9
luindex	94.1	1.1	15	2.7
lusearch	99.3	1.0	11	2.8
pmd	98.9	1.0	18	6.1
xalan	97.1	1.1	54	2.8
compress	98.6	1.0	14	0.7
db	99.8	1.0	84	1.1
jack	99.3	1.0	8	1.8
javac	97.5	1.3	1,843	5.6
jess	49.2	1.9	654	1.8
mtrt	78.5	4.2	21	1.4
raytrace	100.0	1.0	1	2.3
beautyj	84.7	1.4	8,919	8.1
findbugs	96.0	1.3	15,052	11.2
jaranalyzer	92.8	1.1	1,643	2.3
javaguard	52.5	3.0	10	3.2
jdepend	100.0	1.0	1	1.4
psjbb	99.6	1.0	6,686	1.8
volano	99.1	1.0	52	3.8
summary	91.3	1.4	15,052	3.9

In Column 2, we show the percentage of live objects with exactly one referent, averaged across the heap sizes. Next, in Column 3, we report the average number of incoming pointers for live objects across the heap sizes. Column 4 presents the maximum number of incoming pointers across the heap sizes (we report the maximum of the maxima that we observe). The last column shows the space overhead imposed by SRC relative to the heap size (we report average across the heap sizes). This space overhead depends on the object graph (the number of objects with more than one referent) and the number of unaligned root references.

5.3 Pause Times

Tables VI through VIII compare average and maximum pause times for Lisp2 and SRC across the benchmarks. Subsequent tables correspond to the 0%, 5%, and 10% dead space ratio, respectively. We report average values that we have obtained across the heap sizes. For the 5% dead space ratio, SRC reduces maximum pause times by up to 23% and by 12%, on average, and decreases average pause times by up to 23% and by 13%, on average. The biggest performance impact is seen in benchmarks in which the dense prefix tends to be large (objects do not die at the beginning of the heap) and few objects have more than one referent. Higher dead space ratio results in shorter pause times, but within the range between 0% and 10% pause time reduction does not change significantly.

Figure 2 shows average GC pause times for Lisp2 and SRC across the heap sizes for selected benchmarks (representative of our benchmark set). We

Table VI. Performance Data Obtained for SRC and Lisp2 for the Dead Space Ratio in the Old Generation Set to 0%

Benchmark Program	Throughput Increase [%]	Avg. Pause Decrease [%]	Max. Pause Decrease [%]	Nonmoving Objects [%]
raytrace	14.8	18.7	15.7	98.6
psjbb	12.6	19.3	14.8	84.7
fop	11.5	11.6	11.4	65.4
findbugs	9.7	13.5	11.9	32.3
hsqldb	9.2	13.2	13.5	1.6
javac	8.1	12.1	12.8	37.3
beautyj	7.0	12.2	12.2	12.3
mtrt	5.6	20.4	15.8	83.3
average	9.1	15.7	14.0	50.7
jdepend	3.1	20.5	18.3	41.2
db	2.7	20.1	18.1	23.5
pmd	2.5	9.3	8.7	80.4
volano	2.2	9.7	7.5	78.9
javaguard	1.7	12.6	8.1	85.6
lusearch	1.4	10.9	12.2	15.6
eclipse	1.3	10.7	10.3	21.5
xalan	1.1	10.1	10.0	23.6
chart	0.7	13.5	14.0	16.0
luindex	0.6	9.6	9.3	24.7
antlr	0.5	7.6	8.1	33.1
jaranalyzer	0.4	12.5	14.6	17.7
compress	0.3	5.9	4.5	78.4
jack	0.2	7.9	6.6	15.2
jython	0.1	10.6	12.4	7.5
jess	−0.1	8.6	9.5	1.9
bloat	−0.3	8.8	8.9	10.5
average	3.9	12.4	11.6	39.6

In Column 2, we report throughput increase due to SRC (relative to Lisp2) for minimum heap sizes. Benchmarks are listed in descending order according to throughput increase. In Columns 3 and 4, we show the percentage decrease in average (Column 3) and maximum (Column 4) full GC pause times—we compare SRC and Lisp2 across the heap sizes here. The last column presents the percentage of live objects that do not move in SRC (average across the heap sizes). We summarize the eight benchmarks whose execution time benefits most from SRC (the top average row), as well as all benchmarks (the bottom average row).

present the results obtained for the dead space ratio of 5%. SRC consistently reduces full GC pauses, thus increasing minimum mutator utilization (MMU) and consequently improving application throughput.

5.4 Application Throughput

SRC reduces full-heap GC time and thus improves program performance. Throughput impact is proportional to the amount of GC activity, that is, it is the most significant for the minimum heap sizes. In Tables VI through VIII, we report throughput increase due to SRC, relative to Lisp2, for the 0%, 5%, and 10% dead space ratio, and the minimum heap sizes. Average performance improvement across benchmarks is better for higher values of the dead space ratio: 3.9% for 0%, 4.6% for 5%, and 5.5% for 10%. Several benchmarks, however, do not trigger sufficient GC to benefit from SRC in terms of their execution

Table VII. Performance Data Obtained for SRC and Lisp2 for the Dead Space Ratio in the Old Generation Set to 5%

Benchmark Program	Throughput Increase [%]	Avg. Pause Decrease [%]	Max. Pause Decrease [%]	Nonmoving Objects [%]
mtrt	16.6	20.3	15.5	99.1
raytrace	15.2	18.7	18.0	100.0
hsqldb	13.3	19.6	15.9	100.0
findbugs	12.6	15.5	12.8	70.2
fop	11.4	12.8	8.9	100.0
beautyj	9.0	14.2	11.7	51.8
javac	8.3	13.5	14.1	43.0
psjbb	7.3	19.7	14.7	90.2
average	10.8	16.0	13.6	83.2
pmd	3.9	9.7	11.1	94.3
jdepend	3.0	23.4	22.8	100.0
volano	2.8	10.1	8.7	95.6
db	2.6	22.5	19.6	78.2
xalan	2.3	10.1	9.1	60.2
antlr	2.1	7.6	8.2	54.4
eclipse	1.3	10.7	11.4	51.3
javaguard	0.9	12.4	10.5	99.9
chart	0.7	13.4	14.6	25.5
jack	0.5	8.0	7.9	67.1
jython	0.4	10.2	11.6	49.0
jess	0.3	9.0	9.9	82.5
luindex	−0.0	8.8	10.1	44.4
bloat	0.0	9.7	8.2	65.8
jaranalyzer	−0.1	13.0	14.7	36.5
lusearch	−0.2	11.5	11.7	65.4
compress	−0.2	6.0	3.6	81.6
average	4.6	13.2	12.2	72.2

In Column 2, we report throughput increase due to SRC (relative to Lisp2) for minimum heap sizes. Benchmarks are listed in descending order according to throughput increase. In Columns 3 and 4, we show the percentage decrease in average (Column 3) and maximum (Column 4) full GC pause times—we compare SRC and Lisp2 across the heap sizes here. The last column presents the percentage of live objects that do not move in SRC (average across the heap sizes). We summarize the eight benchmarks whose execution time benefits most from SRC (the top average row) as well as all benchmarks (the bottom average row).

time. When we consider the 8 benchmarks with the highest GC activity for each configuration, SRC increases throughput, on average, by 9% for 0%, 11% for 5%, and 13% for 10%.

Figure 3 presents execution time curves for Lisp2 and SRC across the heap sizes for selected benchmarks. The dead space ratio is set to 5%. For these benchmarks, SRC improves performance across the heap sizes (not only for the minimum ones).

5.5 Space Overhead

SRC imposes space overhead because of the overflow table. In the last column in Table V, we report the size of the overflow table, relative to the heap size, for each benchmark across the heap sizes. Space overhead in SRC depends on the object graph and not on the dead space ratio. Space overhead

Table VIII. Performance Data Obtained for SRC and Lisp2 for the Dead Space Ratio in the Old Generation Set to 10%

Benchmark Program	Throughput Increase [%]	Avg. Pause Decrease [%]	Max. Pause Decrease [%]	Nonmoving Objects [%]
mtrt	21.1	20.3	15.2	99.4
raytrace	14.9	18.1	18.0	100.0
psjbb	14.1	19.9	14.0	95.5
fop	13.2	13.0	8.1	100.0
beautyj	12.8	15.9	14.6	61.2
findbugs	12.0	16.2	12.3	78.2
hsqldb	11.6	21.0	16.4	100.0
javac	9.8	13.2	9.8	51.8
average	12.8	16.4	13.1	83.2
xalan	5.5	10.1	9.8	62.4
pmd	3.6	9.7	10.2	94.1
javaguard	3.6	12.6	10.6	100.0
jdepend	3.1	23.4	22.6	100.0
volano	2.7	10.1	9.3	96.9
db	2.6	23.8	19.6	94.4
jess	1.4	9.0	9.9	86.5
luindex	1.3	8.0	8.7	47.7
chart	1.1	13.3	14.3	28.4
antlr	1.0	7.6	8.2	58.9
compress	0.9	6.3	4.8	84.6
bloat	0.9	9.8	9.5	80.3
jack	0.7	8.0	7.9	67.9
eclipse	0.3	9.6	8.8	59.7
jaranalyzer	0.2	12.8	14.3	32.8
jython	0.1	10.4	11.0	54.4
lusearch	-0.5	11.5	12.0	75.5
average	5.5	13.3	12.0	76.4

In Column 2, we report throughput increase due to SRC (relative to Lisp2) for minimum heap sizes. Benchmarks are listed in descending order according to throughput increase. In Columns 3 and 4, we show the percentage decrease in average (Column 3) and maximum (Column 4) full GC pause times—we compare SRC and Lisp2 across the heap sizes here. The last column presents the percentage of live objects that do not move in SRC (average across the heap sizes). We summarize the eight benchmarks whose execution time benefits most from SRC (the top average row), as well as all benchmarks (the bottom average row).

is highest in benchmarks where many pointers end up in the overflow table, that is, those with many objects with multiple referents as well as those with many roots (roots overflow by default due to the lack of word alignment). Thus, space overhead not always fully correlates with the average number of incoming pointers—the number of roots has an impact on the overflow table size too. On average, across benchmarks, space overhead is 3.9%.

As we mentioned previously, SRC can fall back to Lisp2 to guarantee a particular space bound. For these results, however, no fallback is needed, since space overhead is small.

5.6 Dense Prefix

Both Lisp2 and SRC do not compact the prefix of the heap to avoid low-yield yet expensive collections. The size of this prefix is controlled by the dead space

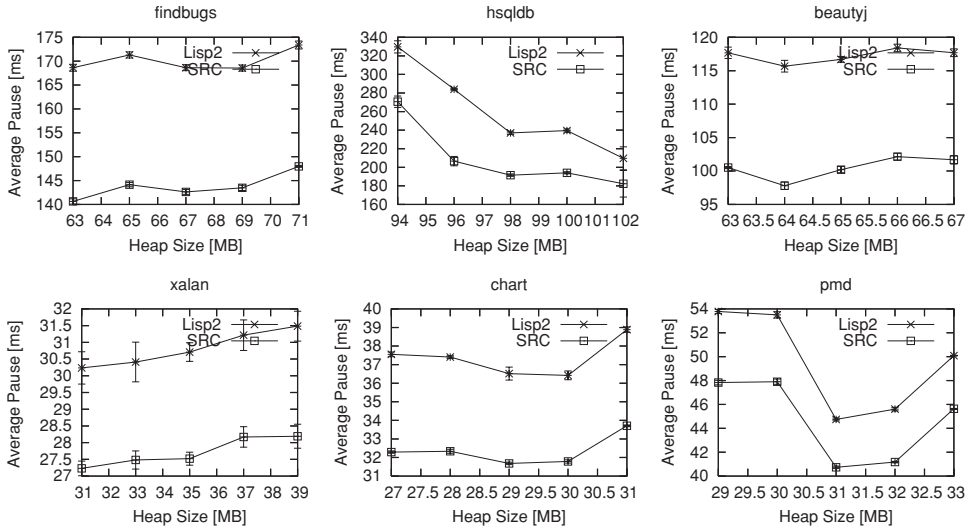

Fig. 2. Full GC pause times in milliseconds for SRC and Lisp2 across the heap sizes for six selected benchmarks. For each benchmark, we report the average values across runs (data points) and standard deviation (error bars).

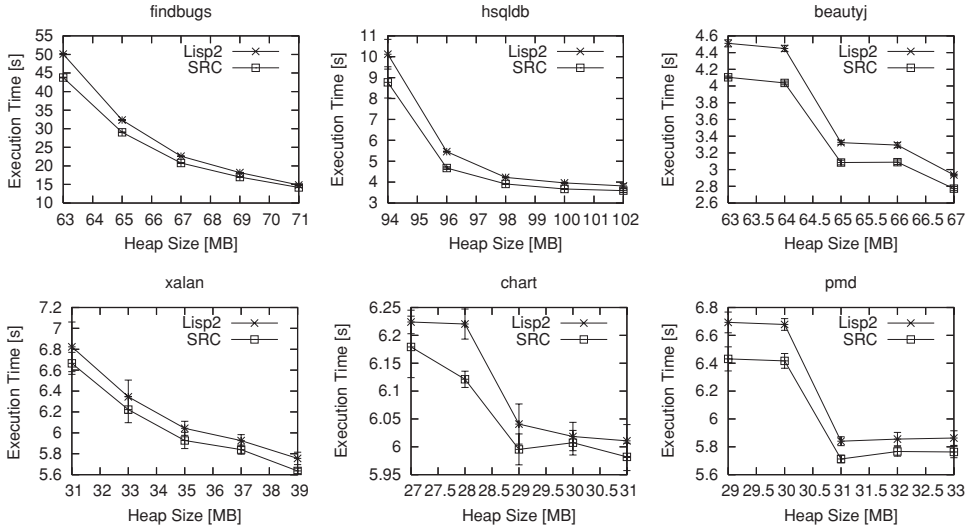

Fig. 3. Execution time in seconds for SRC and Lisp2 across the heap sizes for six selected benchmarks. For each benchmark, we report the average values across runs (data points) and standard deviation (error bars).

ratio. Tables VI through VIII (last column) show the percentage of objects that do not move during compaction for each benchmark. On average, it is 40% for 0%, 72% for 5%, and 76% for 10%. Thus, by default in HotSpot (i.e., for 5%), most objects do not move and most pointers do not need to be processed.

Figure 4 lends insight into how the size of the dense prefix depends on the heap size. We report the percentage of nonmoving live objects as a function

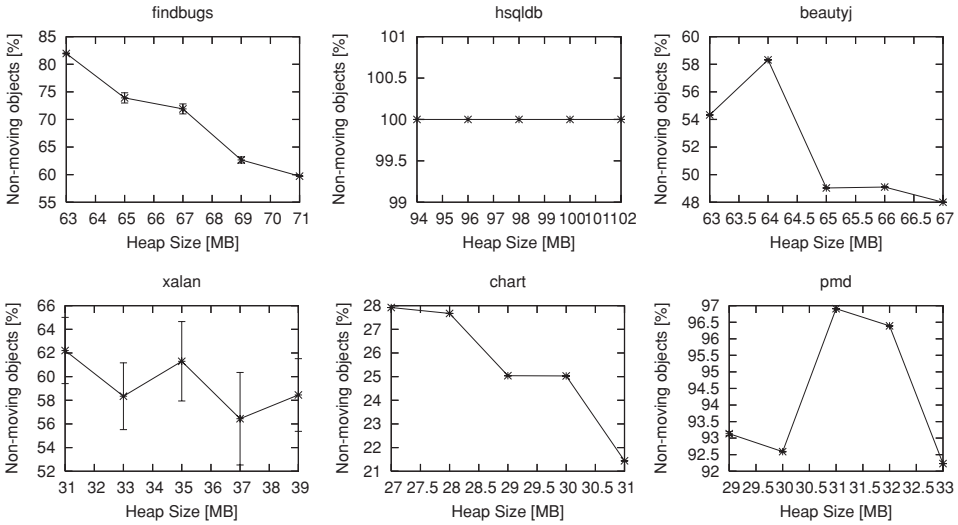

Fig. 4. Percentage of live objects that do not move in SRC as a function of heap size for six selected benchmarks. For each benchmark, we report the average values across collections (data points) and standard deviation (error bars).

of heap size for selected benchmarks. We use the 5% dead space ratio. The percentage of nonmoving objects typically stays the same or decreases as heap size increases. This is because for larger heaps, GC is less frequent and more objects become unreachable between subsequent GCs. As a result, large dead regions are more likely to appear in the heap and necessitate moving.

6. RELATED WORK

SRC employs pointer threading, a technique used by the Jonkers compactor [Jonkers 1979]. The Jonkers algorithm has been shown by Cohen and Nicolau [1983] to introduce potentially significant overhead for pointer manipulation. Lisp2 [Jones 1996], despite comprising four phases, is favored over the three-phase Jonkers GC due to its simplicity and superior performance (e.g., the HotSpot JVM implements Lisp2 instead of Jonkers). SRC has three phases, like the Jonkers GC, but SRC performs few pointer updates, while in the Jonkers algorithm the whole object graph is first reversed and then restored.

The compaction algorithm described in Martin [1982], reverses the object graph during marking, but unlike SRC, does not take into account the single-referent property and does not use the overflow table. The Martin algorithm does not assume dedicated object headers and reserves a bit per pointer for a flag indicating if a pointer is reversed. Space efficiency of the Martin GC depends on the object graph—the algorithm needs additional space for each to-the-right pointer that points to an object that has outgoing pointers. In contrast, SRC's space efficiency depends on the number of incoming pointers. The Martin algorithm always reverses the whole object graph and later restores it while SRC strives to keep pointer manipulation to a minimum.

Since SRC employs stop-the-world collection, it reduces pause times without adding significant complexity to the collector and without negatively impacting throughput. The trade-off between throughput and pause times has also been addressed in Blackburn and McKinley [2003] where the authors introduce a hybrid collector (called URC) that combines copying and reference counting to achieve low pause times without significant performance (throughput) penalties. Unlike SRC, URC does not exploit any behavioral patterns in Java programs and uses reference counting (SRC is a tracing GC).

An alternative approach to reducing pause times is using incremental, concurrent, and parallel collection [Dijkstra et al. 1976; Steele 1975; Yuasa 1990; Doligez and Leroy 1993; Lang and Dupont 1987; Larose and Feeley 1998; Furusou et al. 1991; Detlefs et al. 2004]. Such systems can achieve lower pause times than those that we report herein, but do so (i) for a different application domain than the one we target and (ii) at the cost of both simplicity, application throughput, and memory footprint. Concurrent and real-time GCs target specialized systems with a large number of processors (e.g., servers and cluster systems) and applications with short response time requirements (e.g., hard and soft real-time systems, server applications). Most deskside (client) general-purpose machines today (the domain on which we focus) have a single or small number of processors. Concurrent GC systems (non-stop-the-world collectors) necessarily reduce application throughput since GC and mutator activity are interleaved and must be coordinated. In addition, concurrent collectors often require operating system support to arbitrate conflicts between the mutator and collector threads (e.g., the Compressor [Kermany and Petrank 2006], Pauseless GC [Click et al. 2005], and Mapping Collector [Wegiel and Krintz 2008]).

Recent and important examples of concurrent GC systems include the mostly concurrent collection algorithm [Boehm et al. 1991], its implementations, and extensions. With mostly concurrent GC, marking primarily occurs concurrently with the application execution. The system does so by employing a write barrier to alert the GC to concurrent mutations to the heap by the program. This reduces the pause time required during the stop-the-world phase of the collector, since most marking has been previously completed. The system also reduces application throughput and uses thread priorities to trade off throughput for pause time. Printezis and Detlefs [2000] describe an implementation of this system for the old generation of a generational GC system. The implementation piggy backs on the write barriers required for mostly concurrent collection on those for capturing old-to-young generation references. Ossia et al. [2002] present and implement a mostly concurrent GC for multithreaded, large-scale, server applications that employ very large heaps. The collector combines incremental and priority-aware concurrent GC and achieves effective load balancing of parallel GC threads. On-the-fly GC [Domani et al. 2000; Azatchi et al. 2003], is a generational mark-sweep collector that eliminates the need for a system-wide safepoint, that is, it never stops all mutator threads at the same time. In Levanoni and Petrank [2006], the authors present on-the-fly reference counting GC for multiprocessors.

The concurrent GC systems that are most closely related to ours, however, are those that employ compaction or copying to eliminate heap fragmentation. Ossia et al. [2004] present a mostly concurrent compaction approach for mark-sweep garbage collection to reduce pause times. The system updates references to new locations concurrently and employs incremental compaction, as described earlier. The authors extend incremental compaction so that the GC selects regions to compaction depending on the behavior of the previous mark and sweep phases.

An improvement over earlier mostly concurrent GC techniques [Ossia et al. 2002; Printezis and Detlefs 2000; Boehm et al. 1991] was described in Barabash et al. [2003] where the authors improve application throughput by reducing repetitive GC work by undirtying cards with no traced objects.

Although the GC described in Hosking [2006] is not a mark-compact collector, it performs concurrent mostly copying collection to enable short pause times and to avoid fragmentation. MC^2 [Sachindran et al. 2004] is a incremental copying garbage collector, targeted for memory-constrained devices, which divides the heap into fixed-sized windows and uses write barriers and remembered sets to implement incremental marking and per-window evacuation. In contrast, SRC is a sliding mark-compact GC.

The Compressor [Kermany and Petrank 2006] is a concurrent, incremental, and parallel compaction algorithm that compacts the heap in two phases (marking and compacting). The Compressor manipulates virtual memory mapping and compacts objects into a separate area in the process address space. The compactor preserves object order while reducing pause times significantly by requiring a single pass of the heap for compaction. In prior work [Wegiel and Krintz 2008], we improve upon this approach by eliminating copying of objects altogether through the use of virtual memory mapping. MarkCopy [Sachindran and Moss 2003] leverages virtual memory mapping to reduce the space overhead of a copying collector—it does not require a copy reserve, since it maps and unmaps consecutive pages as copying progresses. SRC provides a simpler approach to reducing the overhead of compaction without compacting objects using a separate area in the virtual address space and without relying on operating system support for virtual memory manipulation by the JVM (which is more complex and less portable).

Abuaiadh et al. [2004] describe a parallel compaction strategy. The main drawback of this strategy is its fix-up phase that must iterate over all pointers, and for each pointer, it needs to perform constant-time, but nontrivial, calculations. SRC does not iterate over pointers that do not need to be changed and for those that do, SRC performs updates without any calculations (via forwarding pointers).

Many other collectors employ multiple threads during garbage collection to spread the GC workload across multiple processors [Abuaiadh et al. 2004; Ben-Yitzhak et al. 2002; Kermany and Petrank 2006]. This parallel collection, in general, reduces pause times when multiple processors are available for use by the GC. Our system is amenable to parallelization, and we overview the algorithm for enabling parallel mark-compact in Section 3.

7. CONCLUSIONS

We contribute SRC, a new compactor that exploits two statistical observations to significantly reduce the cost of pointer adjustment. First, most objects in modern programs have exactly one incoming pointer. Second, state-of-the-art MREs do not move a large part of the heap during compaction. The key idea in SRC is to reverse the direction of pointers in the heap without large space overhead by reusing object headers. Pointer reversal is done during marking and enables SRC to avoid processing of all pointers in the heap—SRC iterates only over pointers that change. SRC has three phases and supports both single-threaded and parallel compaction. We implement SRC in the HotSpot JVM as a full-heap collector that compacts all generations. We evaluate SRC using 25 Java programs in terms of GC pause times and application throughput. In addition, we provide measurement results that support the statistical properties that underlie SRC. The key design feature of SRC is simplicity. Our experiments demonstrate that SRC outperforms the Lisp2 algorithm while introducing modest space overhead, which can be bounded by a fallback to Lisp2.

REFERENCES

ABUAIADH, D., OSSIA, Y., PETRANK, E., AND SILBERSHTEIN, U. 2004. An efficient parallel heap compaction algorithm. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'04)*. ACM, New York.

AZATCHI, H., LEVANONI, Y., PAZ, H., AND PETRANK, E. 2003. An on-the-fly mark and sweep garbage collector based on sliding view. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'03)*. ACM, New York.

BARABASH, K., OSSIA, Y., AND PETRANK, E. 2003. Mostly concurrent garbage collection revisited. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'03)*. ACM, New York.

BEN-YITZHAK, O., GOFT, I., KOLODNER, E., KUIPER, K., AND LEIKEHMAN, V. 2002. An algorithm for parallel incremental compaction. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM'02)*. ACM, New York, 100–105.

BLACKBURN, S., GARNER, R., MCKINLEY, K. S., DIWAN, A., GUYER, S. Z., HOSKING, A., MOSS, J. E. B., AND STEFANOVIC, D. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'06)*. ACM, New York.

BLACKBURN, S. M., GARNER, R., HOFFMANN, C., KHANG, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., ET AL. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'06)*. ACM, New York.

BLACKBURN, S. M. AND MCKINLEY, K. S. 2003. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'04)*. ACM, New York.

BLACKBURN, S. M. AND MCKINLEY, K. S. 2008. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York.

BOEHM, H.-J., DEMERS, A. J., AND SHENKER, S. 1991. Mostly parallel garbage collection. *ACM SIGPLAN Notices 26,* 6, 157–164.

CLICK, C., TENE, G., AND WOLF, M. 2005. The pauseless GC algorithm. In *Proceedings of the International Conference on Virtual Execution Environments*. ACM, New York.

COHEN, J. AND NICOLAU, A. 1983. Comparison of compacting algorithms for garbage collection. *ACM Trans. Program. Lang. Syst. 5,* 4, 532–553.

DETLEFS, D., FLOOD, C., HELLER, S., AND PRINTEZIS, T. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*. ACM, New York.

DEUTSCH, L. P. AND BOBROW, D. G. 1976. An efficient incremental automatic garbage collector. *Comm. ACM 19,* 9, 522–526.

DIECKMANN, S. AND HÖLZLE, U. 1998. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of 12th European Conference on Object-Oriented Programming (ECOOP98)*.Springer-Verlag, Berlin, 92–115.

DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. 1976. On-the-fly garbage collection: An exercise in cooperation. Lecture Notes in Computer Science, vol. 46. Springer-Verlag, Berlin.

DIWAN, A., ED. 2004. *Proceedings of the 4th International Symposium on Memory Management*. ACM Press.

DOLIGEZ, D. AND LEROY, X. 1993. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 113–123.

DOMANI, T., KOLODNER, E., AND PETRANK, E. 2000. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*. ACM, New York.

FISHER, D. A. 1974. Bounded workspace garbage collection in an address order preserving list processing environment. *Inform. Process. Lett. 3,* 1, 25–32.

FLOOD, C., DETLEFS, D., SHAVIT, N., AND ZHANG, C. 2001. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*.

FURUSOU, S., MATSUOKA, S., AND YONEZAWA, A. 1991. Parallel conservative garbage collection with fast allocation. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'91)*. ACM, New York.

GPLJAVA. Open Source Java Software. http://java-source.net.

HOSKING, A. L. 2006. Portable, mostly-concurrent and mostly-copying garbage collection for multiprocessors. In *Proceedings of the 4th International Symposium on Memory Management (ISMM'06)*. ACM, New York, 40–51.

HOTSPOT JVM GC. HotSpot JVM GC. http://java.sun.com/javase/technologies/hotspot/gc/index.jsp.

JONES, R. E. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York. (With a chapter on Distributed Garbage Collection by R. Lins.)

JONKERS, H. B. M. 1979. A fast garbage compaction algorithm. *Inform. Process. Lett. 9,* 1, 25–30.

KERMANY, H. AND PETRANK, E. 2006. The Compressor: Concurrent, incremental and parallel compaction. In *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*. ACM, New York, 354–363.

LANG, B. AND DUPONT, F. 1987. Incremental incrementally compacting garbage collection. In *Proceedings of the Symposium on Interpreters and Interpretive Techniques (SIGPLAN'87)*. ACM, New York, 253–263.

LAROSE, M. AND FEELEY, M. 1998. A compacting incremental collector and its performance in a production quality compiler. In *Proceedings of the 1st International Symposium on Memory Management (ISMM'98)*.ACM, New York, 1–9.

LEVANONI, Y. AND PETRANK, E. 2006. An on-the-fly reference counting garbage collector for Java. *ACM Trans. Program. Lang. Syst. 28,* 1.

LIEBERMAN, H. AND HEWITT, C. E. 1981. A real-time garbage collector based on the lifetimes of objects. AI Memo 569a, MIT, Cambridge, MA.

MARTIN, J. J. 1982. An efficient garbage compaction algorithm. *Comm. ACM 25,* 8, 571–581.

MORRIS, F. L. 1978. A time- and space-efficient garbage compaction algorithm. *Comm. ACM 21,* 8, 662–5.

OPENJDK. Open Source J2SE. http://openjdk.java.net.

OSSIA, Y., BEN-YITZHAK, O., GOFT, I., KOLODNER, E. K., LEIKEHMAN, V., AND OWSHANKO, A. 2002. A parallel, incremental and concurrent GC for servers. In *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation*. ACM, New York, 129–140.

OSSIA, Y., BEN-YITZHAK, O., AND SEGAL, M. 2004. Mostly concurrent compaction for mark-sweep GC. In *Proceedings of the 4th International Symposium on Memory Management (ISMM'04)*. ACM, New York.

PRINTEZIS, T. AND DETLEFS, D. 2000. A generational mostly-concurrent garbage collector. In *Proceedings of the 2nd International Symposium on Memory Management (ISMM'00).* ACM, New York.

SACHINDRAN, N. AND MOSS, E. 2003. MarkCopy: Fast copying GC with less space overhead. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'03)*. ACM, New York.

SACHINDRAN, N., MOSS, J. E. B., AND BERGER, E. D. 2004. MC2: High-performance garbage collection for memory-constrained environments. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'04)*. ACM, New York.

SPEC. SPEC Java Benchmarks. http://www.spec.org.

STEELE, G. L. 1975. Multiprocessing compactifying garbage collection. *Comm. ACM 18,* 9, 495–508.

VOLANOMARK. The VolanoMark Benchmark. http://www.volano.com/benchmarks.html

WEGIEL, M. AND KRINTZ, C. 2008. The Mapping Collector: Virtual memory support for generational, parallel, and concurrent compaction. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*. ACM, New York.

WEGIEL, M. AND KRINTZ, C. 2009. Dynamic prediction of collection yield for managed runtimes. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*. ACM, New York.

WILSON, P. R. 1992. Uniprocessor garbage collection techniques. In *Proceedings of International Workshop on Memory Management.* Springer-Verlag, Berlin.

YUASA, T. 1990. Real-time garbage collection on general-purpose machines. *J. Syst. Softw. 11,* 3, 181–198.

Received April 2008; revised September 2008, January 2009, April 2009; accepted June 2009