The Compressor: Concurrent, Incremental, and Parallel Compaction *

Haim Kermany Erez Petrank

Dept. of Computer Science
Technion - Israel Institute of Technology
Haifa 32000, Israel
{haimk,erez}@cs.technion.ac.il

Abstract

The widely used Mark-and-Sweep garbage collector has a drawback in that it does not move objects during collection. As a result, large long-running realistic applications, such as Web application servers, frequently face the fragmentation problem. To eliminate fragmentation, a heap compaction is run periodically. However, compaction typically imposes very long undesirable pauses in the application. While efficient concurrent collectors are ubiquitous in production runtime systems (such as JVMs), an efficient non-intrusive compactor is still missing.

In this paper we present the *Compressor*, a novel compaction algorithm that is concurrent, parallel, and incremental. The Compressor compacts the entire heap to a single condensed area, while preserving the objects' order, but reduces pause times significantly, thereby allowing acceptable runs on large heaps. Furthermore, the Compressor is the first compactor that requires only a single heap pass. As such, it is the most efficient compactors known today, even when run in a parallel Stop-the-World manner (i.e., when the program threads are halted). Thus, to the best of our knowledge, the Compressor is the most efficient compactor known today. The Compressor was implemented on a Jikes Research RVM and we provide measurements demonstrating its qualities.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Languages, Performance, Algorithms.

Keywords Runtime systems, Memory management, Compaction, Garbage collection, Concurrent garbage collection.

1. Introduction

Today's SMP machines that run modern applications using large heaps present new challenges in designing suitable garbage collectors. In particular, modern servers are required to operate continuously and remain highly responsive to extremely frequent client requests and very large heaps. To allow non-intrusive garbage collection, concurrent collectors have been designed and implemented in various modern runtime systems. Concurrent collectors run in parallel to the application on a separate thread using part of the overall computing resources, while the application continues to run on the rest of these resources.

With the exception of the Sapphire collector [14], concurrent garbage collectors do not move objects during collection. In other words, the garbage collector only reclaims unreachable objects. Consequently, the heap may become more and more fragmented as holes are created in between the objects that survive the collections. These holes make allocation more costly. Furthermore, the application may fail in allocating a large object even though the heap does possesses the overall free space required for the requested allocation.

Compaction is used to solve this problem by grouping live objects together in the heap and freeing up large contiguous spaces available thereafter for future allocation. However, known compaction algorithms execute all, or a substantial part of the compaction while the application is halted, freezing the application for a large amount of time. Compaction is notorious for imposing lengthy pause times. Thus, even if the runtime system employs concurrent garbage collectors, eventually a compaction may be triggered causing infrequent, yet extended, pauses.

In this paper, we present the design and implementation of the Compressor: a new efficient compaction algorithm that requires only short pauses and is suitable for Java and C# running on modern SMPs and supporting large heaps. The Compressor is incremental (i.e., application threads contribute some compaction work during each allocation or mutation), parallel (i.e., compaction work can run in parallel on multiple processors), and concurrent (i.e., parts or all of the collection can run concurrently while the application is executing). In addition to reducing pause times, the Compressor is the first compactor that requires only one heap pass while achieving full compaction. Namely, it compacts the entire heap to a single packed area and preserves the order of allocated objects. All known compactors require at least two passes. Therefore, a parallel version of the Compressor, designed to work while the program is halted (in a Stop-the-World manner) is, to the best of our knowledge, the most efficient compactor known today.

1.1 Technique used

To achieve concurrency, the Compressor makes use of the system's page protection mechanisms as in some previous work [3, 4, 10, 22]. In the beginning, the application threads are directed to work with objects as if they had already moved. However, when they try to access objects not yet moved, a trap is sprung and an area surrounding the attempted access location is moved. Its pointers

* This research was supported by Intel Corporation.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

are updated to point to the new locations of the objects' referents. More details appear in Section 2 below.

For a more detailed introduction to garbage collection and memory management, the reader is referred to the book by Jones [15].

1.2 Organization

An overview of the Compressor design is provided in Section 2. The design details are given in Section 3. The implementation is described in Section 4 and the measurements are presented in Section 5. Related work is discussed in Section 6 and we conclude in Section 7.

2. An overview of the Compressor design

We start with an overview of the compactor. A detailed description appears in Section 3 below. The Compressor does not actually compact the heap into itself. Instead, akin to a copying collector, it compacts the heap into a second space. Yet, unlike copying collectors, the Compressor preserves the order of objects in the heap and does not require the safeguarding of a large space for the collection. This desirable behavior is achieved through the use of (standard) virtual memory operations. The Compressor also satisfies the following useful property. After compacting a page (or several pages), it is able to return this page (or pages) to the operating system. Thus, during compaction the algorithm repeatedly allocates new virtual pages, but at the same time returns the same number of (or more) pages to the operating system.

The Compressor assumes as input a standard *markbit* vector that is typically output by any marking procedure. For example, the state-of-the-art On-the-Fly marking procedure presented by Azatchi et al. [5] can be used to produce a *markbit* vector on-the-fly. This vector has one bit for each heap word,[1] and for each live object in the heap, the two bits representing the first and last words of the live object are set.

A small auxiliary *offset* table in the spirit of the IBM's compactor [1] is used to compute the relocation function. Namely, given an address of an object in the heap, this table can be used to compute the new address to which the object is moved. This table is first computed based on the *markbit* vector. This computation can be executed concurrently and without accessing the actual objects in the heap.

We start with the simpler, parallel version of the Compressor. In this version, several compaction threads may move and update pointers in parallel. Each Compressor thread finds a page that has not yet been moved. It moves the objects on this page to their new location according to the new addresses computed from the *offset* table. A new virtual page may be allocated at this point to accept the moved objects. Next, the thread traverses the objects it has moved and fixes their pointers using the *offset* table again. The pages from which the objects were moved can then be safely returned to the operating system. In practice, the size of an area handled by a thread may be (much) larger than a single page.

To obtain a concurrent compactor, we use traps to protect the virtual space into which the objects are moved. After computing the *offset* table concurrently, the above-mentioned virtual space is protected and the roots are updated to point to the target locations (using the *offset* table). Next, a concurrent thread moves the pages of the heap while the program threads provide some help via execution of traps. A program thread trying to access a not-yet-moved object springs the trap, which moves the object and its adjacent objects to their target location in order to fill a target page. The trap also updates the pointers of the moved objects.

Note that each page of the heap is touched only once by the Compressor, whether parallel or concurrent, which is what makes it highly efficient. Typically, a stop-the-world garbage collector is more efficient than a concurrent one, but imposes longer pauses. However, the concurrent Compressor has an important virtue that makes it more efficient: it is cache conscious. The concurrent Compressor is cache conscious because a page is copied only when the objects on this page are needed by the program. Thus, the operation of the Compressor is highly coordinated with the operation of the program with respect to locality. We now provide the details of the algorithm.

[1] Assuming objects are word-aligned.

3. The design details

The details of the algorithm are given in the sections below.

3.1 Nomenclature, data structures, and virtual memory operations

The Compressor uses two virtual address spaces, each of them of the same size as the heap. We call these spaces *virtual* since they are not always mapped to physical addresses. During each compaction objects will be moved from one virtual space (denoted *from-virtual-space*) to the other virtual space (denoted *to-virtual-space*) and their roles will change thereafter.

Four major data structures are used. First, the Compressor assumes a *markbit* vector that is output by the garbage collector's marking phase. This vector has one bit for each word in *from-virtual-space*. Assuming objects are word aligned, this vector will hold set bits corresponding to the first and last words of each live object. Second, the Compressor employs an *offset* vector. We divide the heap (or actually the *from-virtual-space*) into blocks. The size of a block is a parameter that is typically set to 512 bytes. For each such block, there is an entry in the *offset* vector holding a pointer to the location in *to-virtual-space* to which the first object (that start) in the block is moved. Finally, we use two smaller tables, denoted the *first-object* vector and the *status-table*. The *first-object* vector has a pointer for each page P of *to-virtual-space* (typically a page size is 4096 bytes) referencing the location in *from-virtual-space* of the first object that is moved into Page P by the Compressor. The *status-table* has a state for each *to-virtual-space* page. The size of a state should be the minimum size that can be used with a cmp-and-swap operation, typically a word or a byte. The *status-table* signifies for each *to-virtual-space* page whether it is UNHANDLED, i.e., has not been moved to yet, or BUSY, meaning that it is currently being moved to, or HANDLED, meaning that all its objects were already moved and their pointers updated.

The Compressor will use the following virtual memory services widely available on standard operating systems (we used Linux). These services and some of their usage are further described by Appel and Li [4].

Map: Map a virtual page to a physical page.

UnMap: Unmap a virtual page from its associated physical page.

ProtN: Protect a range of virtual pages from read and write access.

UnProt: Remove the protection from a virtual page.

TRAP: Perform a specified routine upon access to a protected virtual page.

DoubleMap: Map one physical page to two different virtual pages.[2]

The double mapping in the above last item is used when working with a protected page during trap handling. While the page remains protected from access by the program threads, the Compressor

[2] This service is actually implemented using the Linux system calls shmget() and shmat().

```
Procedure Get-New-Address(old: address)
begin
1.      blockNumber := Get-Block-Number(old)
2.      blockAddress := Get-Block-Address(old)
3.      offsetInBlock := Total-Live-Data(blockAddress,old)
4.      newAddress :=to-virtual-space+
            offsetVector[blockNumber] + offsetInBlock
end
```

Figure 1. Get-New-Address() Procedure

will touch objects on it by using a second virtual view that is not protected.

3.2 The parallel stop-the-world Compressor

We start by describing the details of the simpler parallel compactor. The more involved Compressor, which is also incremental and concurrent, is described later in Section 3.3.

The Compressor starts by computing the addresses to which objects move, recording this information succinctly. The order of objects in *from-virtual-space* is preserved during the move to *to-virtual-space*. Thus, the new address of an object O is the bottom address of *to-virtual-space* plus the total accumulated size of the live objects residing there before the object O from *from-virtual-space* arrived. We note that the *markbit* vector has the required information to compute the new address for each object, and so a heap pass is not necessary at this stage. In order to facilitate fast computation of a new address for a given object, the Compressor prepares helpful information in the *offset* vector. Using a single pass over the *markbit* vector, the Compressor computes, for each block B, the sum of live space before it, and stores this number in the *offset* vector. This is exactly the location in *to-virtual-space* into which the first live object in the block B is moved.

A second computation executed during the same pass over the *markbit* vector is one that finds, for each *to-virtual-space* page P, the *from-virtual-space* location of the first object that moves into P during the compaction. This information is obtained in the same (single) pass over the *markbit* vector and is stored in the *first-object* vector.

We stress that the use of a forwarding pointer is avoided and this process does not access the actual objects in the heap. Only the small *offset* and *markbit* vectors are touched. The pseudo-code for using the *offset* vector to translate an old address into a new one (i.e., determine the address to which a given object should be moved) appears in Figure 1. This procedure first assumes a macro `Get-Block-Number` that returns the block number of a given address (using a shift and a subtraction), a second macro `Get-Block-Address` computing the beginning address of the block that contains a given address (by zeroing the address' least bits), and a final macro `Total-Live-Data`, which computes the size of live space in the given block between its beginning and the given address. This macro uses the *markbit* vector to obtain the required information.

Figure 2 depicts an example of how the updated new address of a *from-virtual-space* object is computed. In this example the base address of *from-virtual-space* is 1000. The block size is 100 bytes. Objects 2 and 4 are 50 bytes each, object 6 is 75 bytes and the other objects are 25 bytes each. The *markbit* vector has a bit set for the beginning and end of each object. For every block B, the entry at the *offset* vector of B amounts to the total size of the live objects until B. To get the new address of object 8, whose old address is 1575, we first obtain the block number (block number 5, start counting from 0) by rounding down the value of (1575 - 1000)/100. The address of the block is calculated by zeroing the least two digits of 1575 to 1500. Using the *markbit* vector, we calculate the total

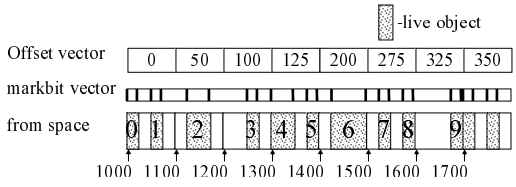

Figure 2. An example: Getting the new address of an object.

```
Procedure Handle-Move-To-Page(pageNumber: int)
begin
1.      pageAddress := Get-Page-Address(pageNumber)
2.      map(pageAddress,PAGE-SIZE)
3.      F := first-object[pageNumber]
4.      L := first-object[pageNumber+1]
5.      O := F
6.      to := Get-New-Address(O)
7.      while O < L do
8.          Fix-References(O)
9.          Copy(O,to)
10.         to := to + size-of(O)
11.         O := Get-Next-Object(O)
12.     unmap(F, L - F)
end
```

Figure 3. Handle-Move-To-Page() Procedure

size of live objects inside the block, i.e., from 1500 to 1575. This amounts to the size of object number 7, which is 25 bytes. These 25 bytes are added to the entry of the block at the *offset* vector which is 275 bytes and we get that the total data until object 8 becomes 300. To get the new address of the object we add this 300 to the bottom address of *to-virtual-space*.

After completing the preparation of the *offset* vector and the *first-object* vector, the actual compaction may begin. Objects are moved page by page according to the target pages at *to-virtual-space*, and their pointers are immediately updated to point to the new locations. These two vectors provide all the information required for the move after which the pointers are conveniently updated.

The compaction itself starts by updating the roots. The job of moving and fixing the rest of the objects is partitioned into tasks. Each task is associated with one or more *to-virtual-space* pages and consists of moving objects into the associated *to-virtual-space* pages and updating the pointers of the moved objects to point to the new locations of their referents. Note that the objects in a task are determined by the target location in *to-virtual-space* and not by the original location. Each compaction thread undertakes a task upon itself via synchronization with the other Compressor threads. Compaction tasks can be executed in parallel with no synchronization. To move objects into a given *to-virtual-space* page P, the Compressor starts by mapping a new physical page to the *to-virtual-space* page P. The objects are then moved starting at the first object specified by the *first-object* vector for Page P. The first location is computed using the *offset* vector and the rest of the objects follow thereafter. Then, their references are updated using the *offset* vector. Finally, the virtual pages in *from-virtual-space* whose objects were moved are unmapped.[3] Using this technique, alternating pages are

[3] In order to determine whether a *from-virtual-space* page can be unmapped, the Compressor needs to know that all the data in the *from-virtual-space* page has been copied already. This can be determined by checking the *status-table* entries of the *to-virtual-space* pages residing just before and/or after the said page.

```
Procedure Parallel-Compact
begin
1.      stop-Program-Threads
2.      Calculate-Offset-Vector-and-First-Object-Vector
3.      for every root r do
4.          r := Get-New-Address(r)
5.      Spawn compacting threads to perform:
6.          P := Get-Next-Unhandeled-Page
7.          while P != NULL do
8.              Handle-Move-To-Page(P)
9.              P := Get-Next-Unhandeled-Page
10.     If all threads done then Resume-Mutators
end
```

Figure 4. Parallel-Compact() Procedure

mapped and unmapped, ensuring that no physical memory overhead is required for the compaction.[4]

The pseudo-code for handling a *to-virtual-space* page appears in Figure 3. The code uses the `Fix-Reference(O)` procedure that finds the pointers in object O and replaces them with the new locations of the referents using the `Get-NewAddress` procedure described above. The `Get-Next-Object(O)` procedure finds the next live object after O using the *markbit* vector.

A delicate point that should be addressed is the handling of objects that begin on one *to-virtual-space* page and end on another. For the parallel algorithm we may arbitrarily decide that an object is associated with the *to-virtual-space* page on which it begins and work with this assumption. We will need a more involved treatment for the concurrent collector which is handled in Section 3.3 below.

The pseudo-code for the parallel Compressor appears in Figure 4. All program threads are halted and the two vectors are computed. Since this operation is fast, we did not bother parallelizing it, although such parallelization can be done using simple tricks. Parallelization of the main work is straightforward. We use several compaction threads, one for each available processor, and let each obtain a task and execute it. The pseudo-code equates a task with a single page, but, of course, a task may consist of several pages needing to be fixed. The `Get-Next-Unhandled-Page` procedure dispatches tasks to the compaction thread in a synchronized manner. The standard trade-off between balancing the work and minimizing synchronization needs to be fine-tuned here: larger tasks mean less synchronization during dispatching but also less work balance, and vice versa.

An important property of this algorithm is that it requires only one heap pass. Previous algorithms required (at least) two heap passes for the compaction. Typically, one pass was used to move the object and another pass was used to adjust the references so that they point to the new locations. The pre-computation of the *offset* vector gives the algorithm its main efficiency advantage. Both the move and the pointer updates can be executed during the same heap pass. This simple idea has never been used before. Note that this idea may also be used with other earlier collectors such as IBM's compactor [1].

3.3 A concurrent, incremental, and parallel compactor

In this section, we note that most of the parallel compaction can be run incrementally by the program threads. Depending on the availability of idle processor time, low priority background threads can also execute some compaction work.

[4] We address the situation of worst case behavior in Section 4.1. In that case, the trap routine ensure that pages are unmapped when a worst case scenario is detected.

The two major steps of the compaction algorithm described above can be executed while the application is running. First, it is possible to compute the *offset* vector and the *first-object* vector concurrently with the running threads. Second, using some modifications, we can also move pages and update pointers concurrently with the program run. The program threads need to be stopped only to fix the roots.

Since fixing the roots requires the information in the *offset* vector, we must compute it before stopping the program threads. Thus, the first phase still consists of computing the *offset* and the *first-object* vectors which is done incrementally. Each program thread contributes some computation when allocating a new object. The first problem that arises here is that new objects may be allocated to holes in the heap between live objects. Such allocations interfere with the computation of the *offset* vector, as they change the values that should be output.

In order to prevent interference of newly allocated objects during this computation, new objects are allocated to *to-virtual-space*. Thus, they will not be moved in later stages. However, such objects cannot be completely ignored. Their pointers point to the objects' original locations and we must now update them to point to the objects' new locations. We denote the pages in *to-virtual-space* containing newly created objects as *require-update* pages.

After calculating the *offset* and the *first-object* vectors concurrently, we move on to protecting all the virtual *to-virtual-space* pages. Note that the program threads are not using this virtual space at this time, and therefore, the protection of these pages can be set concurrently while the program is running and using the *from-virtual-space*. We then stop the program threads. While the program threads are halted, we update the roots to point to their referents' new locations and we protect the *require-update* pages. Observe that we cannot protect the *require-update* pages concurrently with the program run, because the program threads are using these pages. At this point the program threads are resumed.

From this point and on, the program threads will never access the *from-virtual-space*. The roots only reference addresses in *to-virtual-space*, which is protected, and therefore, traps are bound to be triggered soon thereafter. When a program thread accesses a *to-virtual-space* location and gets trapped, the trap routine moves the appropriate objects from *from-virtual-space* into the *to-virtual-space* page on which the trap was sprung. The parallel algorithm was designed for moving objects according to their target *to-virtual-space* pages. Therefore, the trap operation is readily available to us. However, we must distinguish two cases. Traps on *require-update* pages do not move objects, but only update references. Traps on the rest of the *to-virtual-space* pages move the objects and update references as before. After handling a *to-virtual-space* page, we can unprotect it and the program threads can go on working with unprotected pages. Our invariant of letting the program threads access only pointers in *to-virtual-space* holds since these pointers, which the program can read, are already updated and point to *to-virtual-space* only.

While executing the trap, the program threads use the DoubleMap virtual memory primitive in order to access the protected page. A second virtual page is mapped to the same physical page but is not protected and may be used by the trap code to read and write to the protected *to-virtual-space* page. Other program threads that try to access the same *to-virtual-space* page while it is being handled by a trap, will still be trapped. These threads will wait (and yield the processor) until the first trap finishes and the protection of the page is lifted.

To coordinate the handling of pages, the Compressor threads use an additional structure denoted the *status-table*. The *status-*

```
Procedure Trap-Routine(A: address)
begin
1.      P := Get-Page-Number(A)
2.      oldStatus :=
            cmp-and-swap(status-table[P],UNHANDLED,BUSY)
3.      if oldStatus = UNHANDLED then
4.          if P is a require-update page then
5.              Fix-Page(P)
6.          else //P is a standard to-virtual-space page
7.              Move-To-Virtual-Page(P)
8.          unprotect(P)
9.          set(statusTable[P],HANDLED)
10.     elseif oldStatus = BUSY then
11.         while test(statusTable[P]) != HANDLED do
12.             wait// yield processor.
end
```

Figure 5. Trap-Routine() Procedure

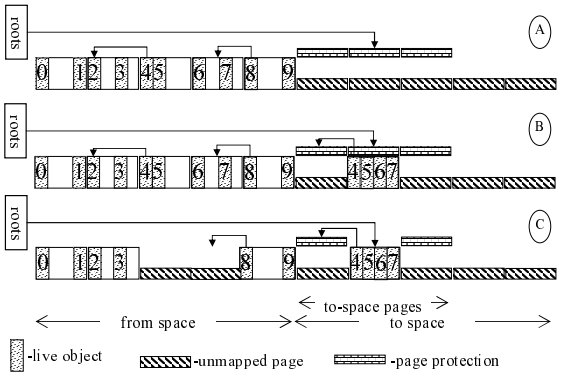

Figure 6. An example: The execution of a trap.

table contains a byte (or word) for each *to-virtual-space* page.[5] The status of all *to-virtual-space* pages is initialized to UNHANDLED. In the beginning of the trap routine, the trap tries to modify the status of the page from UNHANDLED to BUSY using a cmp-and-swap operation. If the status is modified successfully, then the trap handles the page and eventually changes the status to HANDLED using the atomic write. If the cmp-and-swap operation fails, then another mutator is already handling the page and the trap routine just waits until the status of the page is changed by the other thread into HANDLED.

Allocations that occur concurrently with the run of the Compressor are handled as follows. All allocations performed after the mark phase is completed are put in the *require-update* pages, as discussed above. Once the roots are updated, newly created objects do not require pointer updates. From this point and on, we allocate objects in *to-virtual-space* pages that are marked HANDLED. These pages are not protected and are not touched further by the Compressor.

Let us refer again to the delicate issue of objects that stretch along more than one *to-virtual-space* page. In order to remove the protection of a *to-virtual-space* page, we must copy all the data that belongs to this page, even if this data does not consist of complete objects. Indeed, we copy the end of the object that starts on the previous page and the beginning of an object that ends on the next page. To save physical pages (and involved management), we chose not to copy the other parts of these objects, and thus, we do not need to map the adjacent pages until they need to be copied in their entirety.

[5] The size of a *status-table* entry is the minimum entity on which we can invoke a cmp-and-swap synchronization operation.

```
Procedure Concurrent-Compact
begin
1.      Execute the following incrementally and/or concurrently
2.          Calculate-Offset-Vector
3.          protect to-virtual-space pages
4.      stop-Mutators
5.      protect require-update pages
6.      for every root r do
7.          r := Get-New-Address(r)
8.      Resume-Mutators
9.      // Now traps occur on accessed protected pages.
10.     If an idle processor is available, execute concurrently:
11.         P := Get-Next-Unhandeled-Page
12.         while P != NULL do
13.             Handle-Move-To-Page(P)
14.             P := Get-Next-Unhandeled-Page
end
```

Figure 7. Concurrent-Compact() Procedure

The code of the trap routine appears in Figure 5. The cmp-and-swap routine atomically compares the contents of a memory location (the first parameter) to a given value (the second parameter) and, if they are the same, modifies the contents of this memory location to a given new value (the third parameter). It returns the value that existed in the memory location before the operation. In our case, if the returned value is UNHANDLED, then the trap becomes the mover of this page. Otherwise, it executes wait until the other thread handling this page marks it HANDLED.

Figure 6 illustrates the course of a single trap (for simplicity, the first one.) In the example, ten live objects reside in the *from-virtual-space* marked 0 to 9. Objects 0 to 3 should move to the first *to-virtual-space* page, objects 4 to 7 should move to the second *to-virtual-space* page and the objects 8 and 9 should move to the third *to-virtual-space* page. The state of the heap before the trap is depicted in the upper part (State A). Since only three *to-virtual-space* pages will be used, only these three pages are protected. At Stage A no object has been moved yet, but the root has already been fixed and is pointing to the new location of Object 6. When the application tries to touch Object 6, a trap is triggered, invoking the trap routine that moves and fixes the references of all the objects that move to the second *to-virtual-space* page, i.e., objects 4 to 7. Object 4 has a reference to Object 2 and it now points to the new location of Object 2. After moving and fixing the objects (State B), the trap routine unprotects the second *to-virtual-space* page and unmaps the appropriate pages in *from-virtual-space* (State C). Note that at the end of the trap Object 8 (in *from-virtual-space*) is still pointing to the old address of Object 7 despite the fact that Object 7 already moved. This pointer will be updated when Object 8 moves.

The overall operation of the concurrent compaction appears in Figure 7. First, the auxiliary vectors are computed and pages are protected incrementally. Each thread contributes a bit to this computation when it allocates. Furthermore, if an idle processor exists, a concurrent low-priority thread is spawned and it helps in finishing this computation more rapidly. Next, while the program threads are halted, the *require-update* pages are protected and the roots get updated. Then, program threads are resumed and the moving of objects and updating of pointers are carried out incrementally by traps that occur on protected pages. Again, if an idle processor is available, it is used to aid the program threads and finish the compaction more rapidly.

3.4 Special treatment of dense blocks

In some (rather frequent) cases, the objects inside a block are already dense. This usually happens to blocks of older objects that were compacted earlier and remained reachable or blocks that contain only one object. Recall that a typical block size is 512 bytes. For objects in such blocks, the computation of the new address is simpler. It is enough to add a single number Δ to the address of each of these objects to obtain its new location. When such a case is identified (during the preparation of the *offset* vector), the value of Δ is put in the *offset* vector for this block. To identify this special case, the least significant bit of the stored value is set. This method turned out to be highly effective in improving the efficiency of the Compressor. A naive version of this method was proposed in [1], where they either treated all blocks as condensed (and obtained a compacted heap that is not fully compacted) or they did not use this method at all. Another collector that made use of dense areas is the MC^2 collector [24]. This collector divides the heap into windows and avoids copying objects in windows with high density. The windows employed by MC^2 are typically much larger than the Compressor's blocks. Thus, while the MC^2 collector looks for mostly-dense windows, the Compressor may expect to find a large number of perfectly dense blocks.

3.5 More Improvements

Some improvements were added to the basic algorithm to enhance its performance. We list these improvements here.

Moving more than one page When a trap occurs, it imposes some overhead. Thus, it makes sense to move more than one page in each trap. We typically moved eight pages per trap. This optimization also reduces the space overhead of the *status-table* and the *first-object* table.

Double mapping to-virtual-space in the beginning The trap routine needs to touch a protected page without lifting the protection. Thus, a second virtual mapping of the heap is required, which is not protected and is available for use by the trap routine. To reduce the trap time, we DoubleMap the entire *to-virtual-space* to a third virtual space when compaction begins. Of course, this mapping can be run concurrently with the program run.

4. An Implementation for Java

We implemented our algorithm on a Jikes RVM [2], a Java virtual machine, using the Fast-Adaptive compiler of version 2.3.4 upon Linux Red-Hat 7.2. The entire system, including the collector itself, is written in Java (extended with unsafe primitives available only to the Java Virtual Machine implementation to access raw memory).

To mark the live objects before the compaction, we used the Jikes provided mark-and-sweep algorithm with some modifications. First, we modified the segregated-fit allocation scheme to use allocation-caches instead. This was done in order to be able to slide the objects down the heap without worrying about their sizes. Second, we modified the collector to switch the sweep phase with our compaction algorithm when compaction is triggered. Note that the Jikes mark-and-sweep collector is a parallel stop-the-world collector, but a concurrent collector (which does not currently exist in Jikes) could be equally used.

4.1 Space overhead

In our implementation we chose a block size of 512 bytes, and moved eight pages per trap. With this choice, the space overhead for the *offset* vector consists of a single word for each 512 heap bytes, i.e. 1/128. The *first-object* and the *status-table* table require a word per page, contributing an overhead of $2 * 1/1024$ of the heap, which is negligible.

The *markbit* vector is a data structure that is attributed to the garbage collector, but requires another 1/32 of the heap. If an 8-byte alignment is employed by the JVM (such as the IBM JVM), then half the overhead is required for the *markbit* vector.

More space overhead that should be considered is the physical space required during the alternation of mapping *to-virtual-space* pages and unmapping *from-virtual-space* pages during the compaction. Note that if the *from-virtual-space* objects that are moved into the *to-virtual-space* are spread among only two pages and these pages contains more live objects that have not yet been copied, then no pages will be unmapped. Such occurrences add to the space overhead. To avoid this violation, we monitor these occurrences, and when the budget of additional mapped pages surpasses a constant threshold, we let the trap routine move some more pages to ensure full evacuation of *from-virtual-space* pages. Such an occasion has never appeared in practice.

A limitation of our compactor on a 32-bit architecture is that the size of the virtual memory must be large enough to contain three virtual heap spaces. These include *from-virtual-space* *to-virtual-space* and the additional view that allow modifying a protected *to-virtual-space* location. This limitation may create some problem for large heaps on a 32-bit machine, but the problem disappears with the modern 64-bit architectures.

5. Measurements

The large server benchmark that we used was the SPECjbb2000 benchmark [11]. This is probably the more interesting benchmark for the Compressor, employing several program threads and larger heaps. For clients benchmarks we used the SPECjvm98 benchmark suite [11] and the Dacapo benchmark suite [25] version beta051009.

The platform used to run the multithreaded SPECjbb2000 was a 2-way HP workstation xw8000 with a 2.4Hz Intel Pentium III Xeon processor and 2GB of physical memory, running RedHat Linux version 2.4.20-31.9smp. The SPECjvm98 benchmarks and the Dacapo benchmarks were run on a 2.8 GHz Pentium 4 Intel uniprocessor with 512M RAM, running RedHat Linux version 2.6.5-1.

To justify the use of compaction, we chose relatively small heap sizes (for the Jikes RVM), yet, not tiny ones. For SPECjbb2000 we used a heap size of 256MB, for the Dacapo and for the SPECjvm98 we used various heap sizes: 24M for fop and ps; 32M for antlr, bloat, jython and pmd; 48M for hsqldb; 72M for xalan; 16MB for jess, jack, and db; 18MB for mtrt; and 20MB for javac. The Jikes RVM requires larger heaps than other JVMs since it uses the same heap for the JVM runtime data structures, the Compressor data structures (including the tables) and the application. Each reported measurement is the average of five runs.

As noted earlier, in our measurements the size of a block is 512 bytes and the number of pages that are moved on a trap is eight. The compaction was invoked every 10 collections in SPECjbb2000 and every 5 collections on the clients benchmarks. We specifically mention when we deviated from these parameters, usually for measurement used to tune these parameters. The two versions of the Compressor, the parallel stop-the-world and the concurrent, are denoted STW and CON, correspondingly.

Unfortunately, there is no previous compaction algorithm provided with Jikes to which we can compare ours. Comparing it to a standard garbage collector does not seem fair, because compaction is notorious for being much slower than a collection, sometimes by a factor of 10; yet no reports on this factor appear in the literature. To check the efficiency of the Compressor, we compared it against two other collectors that are implemented in the Jikes RVM. The first was the Mark-and-Sweep, denoted *MS*. The second was the generational Appel collector denoted GenMS. We also tried a copy-

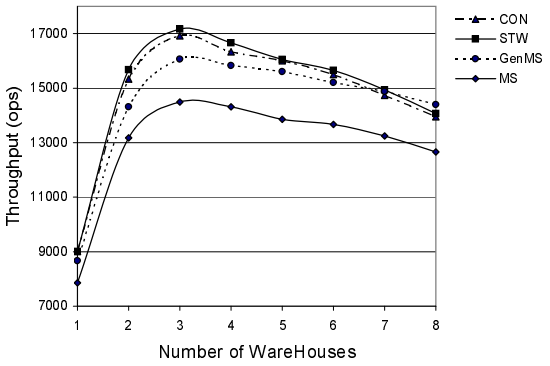

Figure 8. Throughput of Specjbb2000.

	STW	MS	GemMS (nursery only)
jbb 2-WH	319.55	229.37	279.73(29.16)
jbb 4-WH	516.89	287.32	323.64(36.72)
jbb 6-WH	641.53	315.71	347.42(31.17)
jbb 8-WH	770.14	372.46	374.41(22.90)

Table 1. The pause time of the stop-the-world algorithms (ms).

ing collector, but it failed to run on most benchmarks with the tight heap used. The implementations of these collectors are described on [9].

5.1 Server performance

Figure 8 depicts the results running the SPECjbb2000 benchmark over one to eight warehouses and using the four collectors. In this benchmark, the applications starts with one warehouse and increases the number of warehouses to eight, where each warehouse corresponds to an application thread. For each number of warehouses the throughput is checked in operations per second (ops). Higher ops means higher throughput. The better throughput stems from the low efficiency overhead together with the advantage of running with a compacted heap.

Interestingly, the concurrent Compressor provides almost equal, and sometimes even better performance than the stop-the-world parallel Compressor. This can be explained by the cache consciousness of the concurrent Compressor. In particular, objects are moved to their new location in the *to-virtual-space* only when the application accesses them.

In addition to checking the impact of the Compressor on the overall throughput, we also measured the pauses that it imposes. For the parallel Compressor, we measured the time it took to run the compaction while the program was halted. A similar measure was run on the other garbage collectors. The results are presented in Table 1. Traditionally, compaction has been considered a hazard to pause time, as its execution used to take much more time than the execution of a typical (full heap) collection (sometimes by a factor of 10). We can see that the Compressor still takes more time than a typical collection, but its running time is not that far from the collectors measured, and is usually less than a factor of 2. Of course, platforms with more parallel processors will benefit more from its parallelism.

Measuring the behavior of the concurrent Compressor is more problematic. The program threads are interrupted by traps that they execute. We would like to evaluate how much of the CPU time is really used to serve the application and separate it from the time spent on executing the traps. Since the traps are frequent and short, and since the traps start and end in the operating system code, this

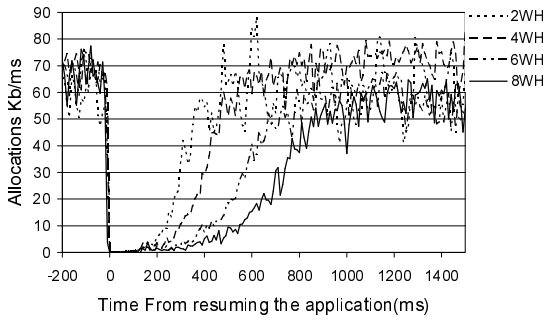

Figure 9. SPECjbb2000: the allocation rate of the program as a function of time.

	3	5	10	15	20	30
jbb2wh	0.987	0.994	1.000	1.006	1.006	0.995
jbb 4wh	0.983	0.980	1.000	1.006	1.010	1.008
jbb 6wh	0.958	0.985	1.000	1.013	1.019	1.014
jbb 8wh	0.949	0.979	1.000	1.012	1.022	1.024
jess	1.126	1.075	1.000	0.953	0.904	0.830
db	1.015	1.005	1.000	1.006	1.001	0.996
javac	1.198	1.100	1.000	0.973	0.990	0.988
mtrt	1.005	1.016	1.000	0.977	0.991	0.961
jack	1.065	1.023	1.000	0.992	0.957	0.953
antlr	1.003	0.991	1.000	0.996	0.998	0.991
bloat	1.090	1.084	1.000	0.936	0.873	n/a
fop	1.300	1.198	1.000	1.004	1.031	1.006
hsqldb	1.279	1.111	1.000	1.007	1.030	0.995
jython	0.968	0.983	1.000	0.983	0.999	0.998
pmd	1.025	1.010	1.000	0.966	0.971	0.999
ps	0.995	0.990	1.000	0.999	0.997	0.988
xalan	0.888	0.948	1.000	1.012	1.024	1.030

Table 2. The performance speedup as a function of the number of collections between compactions.

cannot be easily measured. We chose to measure the application use of the processors by checking the allocation rate as a function of time. Although the rate of allocation is not perfectly stable, it gives a pretty good approximation of the amount of work done by the program. The results are depicted in Figure 9. The X-axis in this figure represents the time (in ms) from the point that the mutators incrementally compute the *offset* and *first-object* vectors. At time 0, the program threads are stopped to fix the roots and protect the *require-update* pages. The program threads resume thereafter, but due to the traps, they do not show full activity until a bit later.

The results in this graph should be compared to Table 1. Namely, we would like to compare the pause that is imposed by a stop-the-world Compressor to the behavior shown in Figure 9. For two warehouses, a pause of 319ms is imposed by a stop-the-world Compressor. Yet the concurrent Compressor allows a noticeable run of the program threads after approximately 200ms. For four warehouses a stop-the-world compaction would take 520ms and the program would start executing after around half that period with the concurrent Compressor.

5.2 Collector characteristics

Tuning compaction triggering Table 2 compares the performance of the benchmarks with various triggering frequencies of the Compressor. The best result of each benchmark is highlighted. In all the benchmarks, excluding SPECjbb2000 and xalan, frequent triggering improves the overall performance of the benchmarks.

Tuning the number of pages that are moved in each trap Since the execution of a trap carries an overhead, the Compressor moves more than one page in each trap. Moving a small number of pages

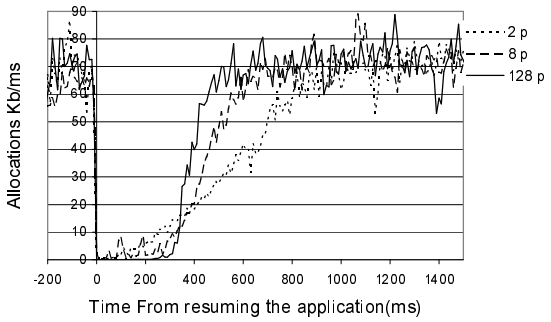

Figure 10. The allocation rate as a function of time, using a different number of pages per trap, in Specjbb2000 with four warehouses.

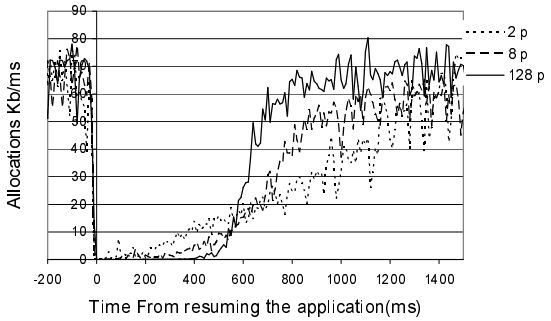

Figure 11. The allocation rate as a function of time, using a different number of pages per trap, in Specjbb2000 with eight warehouses.

during a trap increases the total number of traps but reduces the pause time of each trap. Such a reduction may allow more mutator activity during the concurrent compaction, but the increase in the number of traps will extend the time it takes to finish the compaction and thus, postpone the time where the application may regain full CPU strength. An attempted tuning of this parameter did not noticeably affect the throughput of the program but it did affect the concurrency. In Figures 10 and 11 we can see that when only two pages are moved on a trap, the program receives some CPU share almost right after it resumes, but more time is required until it reaches full activity.

Offset calculation overhead We measured the fraction of time spent on calculating the *offset* vector and the *first-object* vector in the beginning of the collection. It turned out that this part of the compaction took around 5-7% on the SPECjvm98 benchmarks and 11-13% on the SPECjbb2000 benchmark. Since this part of the Compressor execution is short, we did not bother parallelizing its operation and it is run on a single thread in our implementation.

5.3 Client performance

In Figures 12 and 13 the overall running times of the SPECjvm98 and Dacapo benchmarks with various collectors is presented. For these small benchmarks, the Appel generational collector beats the performance of all non-generational collectors. It actually manages to almost refrain completely from running full collections. This phenomenon does not occur at all with the larger SPECjbb2000 benchmark. Concentrating on the non-generational collectors, we first note the similarly to the measurements of the Specjbb2000 benchmarks, the concurrent Compressor yields almost equal, and sometimes even better performance than the stop-the-world parallel

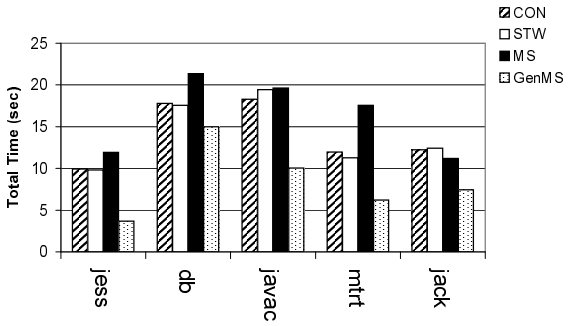

Figure 12. Specjvm98 benchmarks' overall running times with the various collectors.

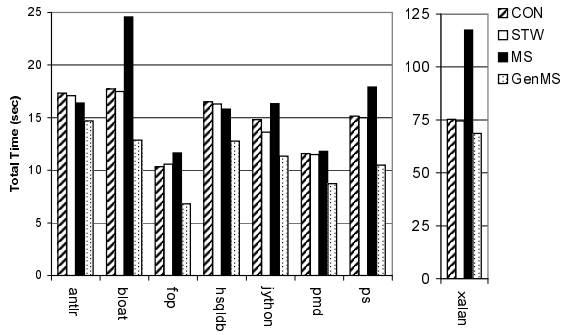

Figure 13. Dacapo benchmarks' overall running times with the various collectors.

Compressor. Second, the Compressor obtains better performance than the MS collector.

The pause times of the Compressor on these benchmarks is depicted again by checking the allocation rate over time in Figures 14, 15 and 16. These graphs should be compared to the stop-the-world pause times presented in Table 3. We can see that though the pause time of the stop-the-word Compressor is relatively high (22-83 ms), with the concurrent Compressor the application does meaningful allocations after a much smaller period of time (typically 5-10 ms).

6. Related work

Compaction algorithms have been known since the 1970s. Older compactors used the simple two-finger technique or the LISP2 algorithm [15]. A more elaborate (and elegant) solution requiring no extra auxiliary data structures is the threaded algorithm of Jonkers [16] and Morris [19]. It was shown in [1] that modern compactors demonstrate a substantial increase in efficiency over these collectors.

Two parallel compactors were presented in [12] and [1]. Flood et al. [12] offered the first parallel compactor. However, their algorithm required three passes over the heap and did not move all objects to a single compacted area. Instead, the heap was split into N areas (where N is the number of processors) and N threads were used to compact the heap into $N/2$ chunks of live objects. An improved algorithm was proposed by Abuaiad et al. [1]. Their algorithm required only two heap passes and offered an almost perfect compaction, in the sense that the resulting heap (after compaction) was almost fully compacted and the order of objects was mostly preserved. The parallel version of the Compressor does better than the above compactors, requiring only a single heap pass, obtaining equally balanced parallelism, and achieving a perfect compaction:

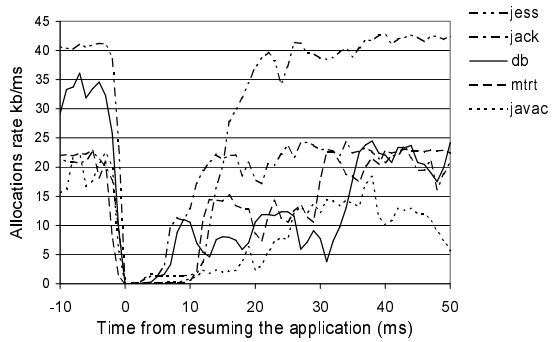

Figure 14. SPECjvm98: The allocation rate of the benchmarks as a function of time.

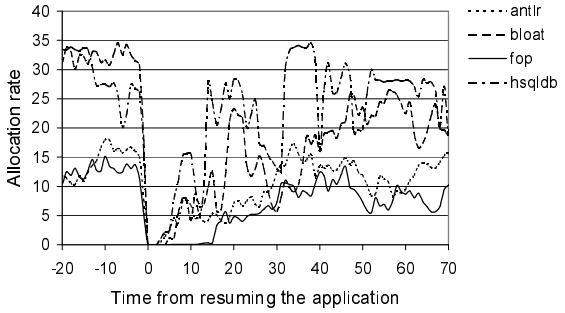

Figure 15. Dacapo: The allocation rate of the benchmarks as a function of time.

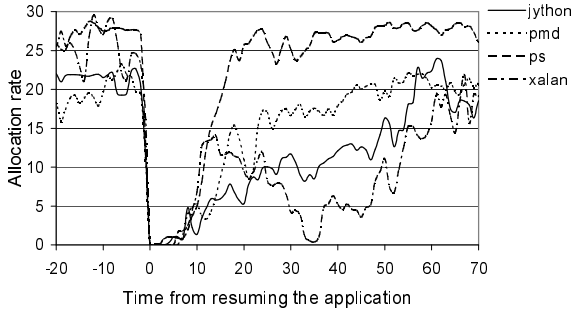

Figure 16. Dacapo: The allocation rate of the benchmarks as a function of time.

	STW	MS	GemMS (nursery only)
jess	30.26	120.20	n/a(5.15)
jack	83.22	115.62	191.79(8.18)
db	67.77	141.51	149.47(5.34)
mtrt	61.31	160.13	156.24(4.49)
javac	67.23	179.59	178.23(8.25)
antlr	29.86	132.26	136.11(4.32)
bloat	75.68	158.48	153.34(6.43)
fop	54.69	194.01	207.32(10.97)
hsqldb	77.61	151.73	223.31(8.95)
jython	41.13	145.82	206.19(9.78)
pmd	47.28	163.48	169.47(5.34)
ps	22.94	123.16	n/a(2.72)
xalan	68.32	177.01	231.15(7.35)

Table 3. The pause time of the stop-the-world algorithms on jvm98 and Dacapo (ms).

the order of objects is perfectly preserved and the resulting heap is perfectly compacted. Thus, the Compressor, as far as we know, is the best parallel compactor existing today.

Incremental compaction was suggested by Lang and Dupont [17] and a modern variant was presented by Ben Yitzhak et al. [8]. The idea was to split the heap into regions, and compact one region at a time by evacuating all objects from the selected region. Extending these works to compact the full heap does not yield the efficient parallel compaction algorithm we need. Extending the first algorithm yields a standard copying collector (that keeps half the heap empty for collection use). Extending the latter is also problematic, since it creates a list of all pointers pointing into the evacuated area; this is not appropriate for the full heap. Also, objects cannot be moved into the evacuated area, since forwarding pointers are kept in place of evacuated objects. Ossia et al. [22] attempted to reduce the compaction pause times by running the pointer updates phase concurrently using virtual memory traps such as ours. However, moving the objects was executed in a stop-the-world manner. They proposed to reduce the pause times further by giving up full compaction and moving only a fraction of the heap objects. The Compressor is more efficient than their compactor as it requires only a single heap pass. Furthermore, the concurrent version of the Compressor runs both the move of the objects and the pointer updates concurrently with the program threads, achieving perfect compaction with shorter pause times.

The Metronome [6] and the bookmarking collector [13] use segregated free lists[6] allocation to achieve compaction with a single objects traversal. However, unlike the compressor, the obtained compaction in these works does not preserve the objects' order and does not compact the objects to a single area in the heap.

Some compaction algorithms (e.g., [18, 6]) use handles to provide an extra level of indirection when accessing objects. Since only the handles need to be updated when an object moves, the fix-up phase, in which pointers are updated to point to the new location, is not required. Nevertheless, a design employing handles is not appropriate for a high performance language runtime system.

Copying collectors obtain compaction for free. However, they differ from compactors because they utilize only half of the heap's space; they move the objects in each collection; and they do not preserve the objects' order. Compaction uses small auxiliary data structures; it may be invoked when necessary; and it preserves the allocation order of objects. The Mark-Copy collector, and its sequel, the MC^2 collector [23, 24] are copying collectors which minimize the additional space required for copying by running a marking phase before the copying begins, and executing the coping incrementally. MC^2 divides the heap into windows, and builds remembered sets for each window during the mark phase. These remembered sets are used to copy each window separately, while each window is copied to the space that the former window evacuated. Using this technic, the additional space required is at most one window size, and the application can resume between the copying one window and the next. Unlike the Compressor, the MC^2 does not preserve the order of objects, its copying phase is not concurrent and it has to scan the roots for each window copying. The Mark-Copy algorithm uses memory services to save memory use in a similar way to the Compressor, but this technique is not used in the sequel MC^2. Two more notable incremental and concurrent copying collectors are the Baker algorithm [7] and the Sapphire [14].

[6] with segregated free lists, memory is divided into fixed-sized pages, and each page is divided into blocks of a particular size. Objects are allocated from the smallest size class that can contain the objects.

7. Conclusions

Mark-and-Sweep garbage collectors suffer from fragmentation, which is handled (infrequently) by compaction algorithms. Compaction executions are notoriously long and impose a high overhead on execution times and extended pauses. Reducing compaction time and its obtrusiveness is an important goal for memory managers today, especially on modern platforms.

In this paper we presented the Compressor: a novel compacting algorithm that requires only a single heap pass. The Compressor is more efficient than previously reported compactors. First, we presented a parallel version of the Compressor that runs efficiently on an SMP utilizing all processors while program threads are halted. Second, we presented a concurrent version of the Compressor that runs incrementally and concurrently, with the program threads achieving high efficiency and shorter pauses. The concurrent collector has high cache consciousness as it moves pages when they are touched by the program threads. Because of this nice cache behavior, its efficiency is not much below, and sometimes may even be higher, than the efficiency of the parallel version, whereas its pauses are much shorter.

The Compressor was implemented on the Jikes Research JVM and measurements demonstrating its efficiency and non intrusiveness were presented.

Acknowledgments

We thank Yoav Ossia, Avi Mendelson, and Harel Paz for helpful discussions. We thank Steve Blackburn and Daniel Frampton for the effort they invested in making the Jikes RVM supportive of compaction. Without their initial work, our implementation work would have been much harder.

References

[1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An efficient parallel heap compaction algorithm. In OOPSLA [21].

[2] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 314–324, Denver, CO, October 1999. ACM Press.

[3] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[4] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. *ACM SIGPLAN Notices*, 26(4):96–107, 1991. Also in SIGARCH Computer Architecture News 19 (2) and SIGOPS Operating Systems Review 25.

[5] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding view. In OOPSLA [20].

[6] David F. Bacon, Perry Cheng, and V.T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, San Diego, CA, June 2003. ACM Press.

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

[8] Ori Ben-Yitzhak, Irit Goft, Elliot Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 100–105, Berlin, June 2002. ACM Press.

[9] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in Java with MMTk. In *ICSE 2004, 26th International Conference on Software Engineering*, Edinburgh, May 2004.

[10] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[11] Spec: The Standard Performance Evaluation Corporation. http://www.spec.org/.

[12] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.

[13] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of SIGPLAN 2005 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Chicago, IL, June 2005. ACM Press.

[14] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE 2001 Conference*, Stanford University, CA, June 2001.

[15] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[16] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):25–30, July 1979.

[17] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In *SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, volume 22(7) of *ACM SIGPLAN Notices*, pages 253–263. ACM Press, 1987.

[18] Martin Larose and Marc Feeley. A compacting incremental collector and its performance in a production quality compiler. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 1–9, Vancouver, October 1998. ACM Press.

[19] F. Lockwood Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–5, 1978.

[20] *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[21] *OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Vancouver, October 2004. ACM Press.

[22] Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. Mostly concurrent compaction for mark-sweep GC. In Amer Diwan, editor, *ISMM'04 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, Vancouver, October 2004. ACM Press.

[23] Narendran Sachindran and Eliot Moss. MarkCopy: Fast copying GC with less space overhead. In OOPSLA [20].

[24] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. MC^2: High-performance garbage collection for memory-constrained environments. In OOPSLA [21].

[25] Dacapo Project: The DaCapo Benchmark Suite. http://www-ali.cs.umass.edu/dacapo/index.html.