An Efficient Parallel Heap Compaction Algorithm

Diab Abuaiadh Yoav Ossia Erez Petrank* Uri Silbershtein
IBM Haifa Research Laboratory
Haifa University, Mount Carmel,
Haifa 31905, Israel.
{diab, yossia,erezp,uris}@il.ibm.com

ABSTRACT

We propose a heap compaction algorithm appropriate for modern computing environments. Our algorithm is targeted at SMP platforms. It demonstrates high scalability when running in parallel but is also extremely efficient when running single-threaded on a uniprocessor. Instead of using the standard forwarding pointer mechanism for updating pointers to moved objects, the algorithm saves information for a pack of objects. It then does a small computation to process this information and determine each object's new location. In addition, using a smart parallel moving strategy, the algorithm achieves (almost) perfect compaction in the lower addresses of the heap, whereas previous algorithms achieved parallelism by compacting within several predetermined segments.

Next, we investigate a method that trades compaction quality for a further reduction in time and space overhead. Finally, we propose a modern version of the *two-finger* compaction algorithm. This algorithm fails, thus, re-validating traditional wisdom asserting that retaining the order of live objects significantly improves the quality of the compaction.

The parallel compaction algorithm was implemented on the IBM production Java Virtual Machine. We provide measurements demonstrating high efficiency and scalability. Subsequently, this algorithm has been incorporated into the IBM production JVM.

Categories and Subject Descriptors

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

General Terms

Languages, Performance, Algorithms

*Dept. of Computer Science, Technion, Haifa 3200, Israel. This work was done while the author was at the IBM Haifa Research Lab. E-mail: erez@cs.technion.ac.il.

Keywords

Garbage collection, Java, JVM, Parallel garbage collection, Compaction, Parallel Compaction.

1. INTRODUCTION

Mark-sweep and reference-counting collectors are susceptible to fragmentation. Because the garbage collector does not move objects during the collection, but only reclaims unreachable objects, the heap becomes more and more fragmented as holes are created in between objects that survive the collections. As allocation within these holes is more costly than allocation in a large contiguous space, program activity is hindered. Furthermore, the program may eventually fail to allocate a large object simply because the free space is not consecutive in memory. To ameliorate this problem, *compaction* algorithms are used to group the live objects and create large contiguous available spaces for allocation.

Originally, compaction was used in systems with tight memory. When compaction was triggered, space had been virtually exhausted and auxiliary space could not be assumed. Some clever algorithms were designed for this setting and could compact the heap without using any space overhead, most notably, the threaded algorithm of Jonkers [9] and Morris [12], also described in [8]. Today, memories are larger and cheaper, and modern computer systems allow (reasonable) auxiliary space usage for improving compaction efficiency. An interesting challenge is to design an efficient and scaleable compaction algorithm appropriate for modern environments, including multi-threaded operating systems and SMP platforms.

In this work, we present a new compaction algorithm that achieves high efficiency, high scalability, almost perfect compaction quality, and low space overhead. Our algorithm demonstrates efficiency by substantially outperforming the threaded algorithm currently employed by the IBM production JVM [6]. It demonstrates high scalability by yielding a speedup factor of about 6.6 on an 8-way SMP. In contrast to previously suggested parallel algorithms, which obtained parallelism by producing several piles of compacted objects within the heap, the proposed algorithm obtains a compaction of high quality by compacting the whole heap to the lower addresses.

Compaction has been notoriously known to cause large pause times. Even with concurrent garbage collectors, a compaction may eventually be triggered causing infrequent, yet, long pauses. Thus, it is important to improve compaction efficiency in order to reduce the maximum pause times. On an 8-way SMP, the new algorithm reduces this pause time by a factor of more than 8 (being more efficient and scalable), thus making the longest pause times bearable.

1.1 Algorithmic overview

A compaction algorithm consists of two parts: moving the objects and fixing up the pointers in the heap to point to the new locations. Let us start with the fix up phase. The simplest fix up method is to keep forwarding pointers in the objects and use them to fix pointers. This implies a space overhead of one pointer per object and forces a read operation in the heap for each pointer update. We use a different approach, dividing the heap into small blocks (typically, 256 bytes) and maintaining fix up information for each block (rather than for each object). This information allows fixing up pointers to live objects within the block. This new approach requires a small amount of computation during each pointer fix up, but restricts the memory accesses to a relatively small block table, rather than reading random locations in the heap.

A second issue is the quality of compaction. We would like to move all live objects to the lower addresses in the heap, even though the work is done in parallel by several threads. The problem is that the naive approach to moving all objects to the lower addresses in parallel requires synchronization for every move operation. Frequent synchronization has a high cost in efficiency. The most prominent previous solution was to split the heap into N areas and let N threads work in parallel, each compacting its own area locally [7]. This method yields several piles of compacted objects. Our goal is to obtain a perfect compaction, moving all live objects to lower addresses as is done in a sequential compaction algorithm. To this end, we divide the heap into many areas. Typically, for a Gigabyte heap, the size of an area is a few megabytes.

Objects in the first few areas are compacted within the area space to lower addresses. This results in a bunch of compacted areas with contiguous empty spaces. The remaining areas are compacted into the free spaces created within areas previously compacted. This allows moving all objects to the lower addresses.

Using the above idea, the quality of compaction does not depend on the number of areas. But now an additional benefit is obtained. Using a large number of areas allows excellent load balancing. Indeed with this improved load balancing, the algorithm scales well to a large number of processors.

1.2 Quality versus overhead

Next, we observe that it is possible to save time and space by somewhat relaxing the compaction quality. Instead of full compaction, we can run a compaction without executing intra-block compaction, but only inter-block compaction. Namely, each block is moved as a whole, without condensing holes within the objects inside it. This allows saving two thirds of the space overhead required by our compaction algorithm. From our measurements of SPECjbb2000, we found it also reduces the (already short) time required to do the compaction by as much as 20%. The area into which the objects are compacted increases by only a few percent due to the holes we do not eliminate.

1.3 Verifying traditional wisdom

It is widely believed that preserving the order of objects during compaction is important. We have made an attempt to design and run a simple compaction algorithm that does not follow this guideline. The proposed algorithm is a modern variant of the *two-finger* algorithm (see [8]) applicable for the Java heap. The advantage of this algorithm is that it substantially reduces the number of moved objects (typically, by a factor of 2). The cost is that it does not preserve the order of objects. The hope was that maybe modern benchmarks are not severely harmed by reordering their objects in this manner. However, it turned out that the folklore belief strongly holds, at least when running the SPECjbb2000 benchmark. Application throughput was drastically reduced with this compaction algorithm.

1.4 Implementation and measurements

We have implemented our algorithms on top of the mark-sweep-compact collector that is part of the IBM production JVM 1.4.0. This collector is described in [5, 6]. The compact part of this collector implements the threaded algorithm of Jonkers [9] and Morris [12]. We have replaced this compactor with our parallel compaction, and ran it on N threads in parallel, using an N-way machine.

In order to compare our algorithm with the original threaded algorithm, we restrict our algorithm to use a single thread. It turns out that our algorithm typically reduces compaction time by 30%. We then check how the compaction scales on a multiprocessor, when run in parallel, and find that it has good scalability, obtaining (on an 8-way AIX machine) a speedup factor of 2.0 on two threads, 3.7 on four threads, 5.2 on six threads, and 6.6 on eight threads. With this new algorithm, compaction ceases to be a major problem on a multiprocessor. For example, using temporary access to a 24-way AIX machine, we ran SPECjbb (from 1 to 48 warehouses) on a 20 GB heap. The last compactions (the most intensive ones) ran for about 15 seconds with the threaded algorithm and only 600 ms with our new parallel algorithm.

1.5 Related work

As far as we know, Flood et al [7] offer the best algorithm known today for parallel compaction. They present an algorithm to compact the (old generation part of the) heap in parallel. Their algorithm uses a designated word per object to keep a forwarding reference. On the HotSpot platform, such a word can be easily obtained in the header of most objects (non-hashed, old-generation objects). The rest of the objects require the use of more space (either free young generation space or more allocated space). The compaction runs three passes over the heap. First, it determines a new location for each object and installs a forwarding pointer, second, it fixes all pointers in the heap to point to the new locations, and finally, it moves all objects. To make the algorithm run in parallel, the heap is split into N areas (where N is the number of processors) and N threads are used to compact the heap into $N/2$ chunks of live objects.

In the algorithm proposed in this paper, we attempt to improve over this algorithm in efficiency, scalability, space consumption, and quality of compaction. In terms of efficiency, our algorithm requires only two (efficient) passes over the heap. Note that we have eliminated the pass that installs forwarding pointers and we have improved the pass that fixes the pointers. In particular, the pass that fixes all pointers in the heap does not need to access the target object (to read a forwarding pointer). Instead, it accesses a small auxiliary table. We also do not assume a header structure that can designate a word to keep the forwarding reference. This reduces space consumption on a platform that does not contain a redundant word per object. In particular, our additional tables consume 2.3% or 3.9% of the heap size (depending on whether the mark bit vector is in use by the garbage collector). Sparing a word per object is likely to require more space [3]. In addition, we improve the load balancing in order to obtain better scalability. We do this by splitting the heap into a large number of small areas, each being handled as a job unit for a compacting thread. This fine load balancing yields excellent speedup. Finally, we improve the quality of compaction (to obtain better application behavior) by moving all live objects to a single pile of objects at the low end of the heap. To do that, instead of compacting each area into its own space, we sometimes allow areas to be compacted into space freed in other areas. Using all these enhancements we are able to obtain improved efficiency, scalability and compaction quality.

Some compaction algorithms (e.g., [11, 2]) 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. However, a design employing handles is not appropriate for a high performance language runtime system.

Incremental compaction was suggested by Lang and Dupont [10] and a modern variant was presented by Ben Yitzhak et al [4]. The idea is 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. We note that our algorithm can be modified to perform incremental compaction. One may determine an area to be compacted and then compact within the area. After the compaction, the entire heap must be traversed to fix pointers, but pointers that do not point into the compacted area can be easily identified and skipped, without incurring a significant overhead.

Finally, the algorithm that we compare against (in the measurements) is the one currently implemented in the IBM production JVM. This is a compaction algorithm based on the threaded algorithm of Jonkers [9] and Morris [12]. These algorithms use elegant algorithmic ideas to completely avoid any space overhead during the compaction. While saving space overhead was a crucial goal at times when memory was small and costly, it is less critical today. Trading efficiency for space saving seems less appealing with today's large memory and heaps. Even though the threaded algorithm requires only two passes over the heap, these passes are not efficient. Our measurements demonstrate that our method shows a substantial improvement in efficiency over the threaded compaction, even on a uniprocessor without parallelization. When SMPs are used, the threaded algorithms do not tend to parallelize easily. We are not aware of any reported parallel threaded compaction algorithm and our own attempts to design parallel extensions of the threaded compaction algorithms do not scale as well as the algorithm presented in this paper.

1.6 Organization

The rest of the paper is organized as follows. In Section 2 we present a sequential version of the new compaction algorithm. In Section 3, we show how to make it parallel. In Section 4, we present a trade-off between the quality of the compaction and the overhead of the algorithm in time and space. In Section 5, we present an adaptation of the two-finger compaction algorithm adequate for Java. In Sections 6 and 7 we present the implementation and the measurements. We conclude in Section 8.

2. THE COMPACTION ALGORITHM

In this section we describe the parallel compaction algorithm. We start with the data structure and then proceed to describe the algorithm.

2.1 Accompanying data structures

Our compaction algorithm requires three vectors: two of them are bit vectors that are large enough to contain a bit for each object in the heap and the third is an offset vector large enough to contain an address for each *block* in the heap. Blocks will be defined later, and the size of a block is a parameter B to be chosen. In our implementation, we chose $B = 256$ bytes.

Many collectors use such two bit vectors and the compaction can share them with the collector. Specifically, such vectors exist in the IBM production JVM collector that we used. The first vector is an input to the compaction algorithm, output by the marking phase of the mark-compact algorithm. This vector has a bit set for any live (reachable) object, and we denote it the *mark bit vector*. The second vector is the *alloc bit vector*, which is the output of the compaction algorithm. Similar to the mark bit vector, the alloc vector has a bit set for each live object after the compaction. Namely, a bit corresponding to an object is set if, after the compaction, the corresponding object is alive.

We remark that in the JVM we use, these vectors exist for collection use. When they do not exist for collector use, these two vectors should be allocated for the compaction. Our garbage collector uses the mark bit vector during the marking phase to mark all reachable objects. The alloc bit vector signifies which objects are allocated (whether reachable or not). This vector is also used during the marking phase since our JVM is non-accurate (i.e., it is not possible to always tell whether a root is a pointer or not). See more on this issue in section 6. Thus, when a pointer candidate is found in the roots, one test the collector runs to check whether it may be a valid pointer, is a check if it points to an allocated object. Here the alloc bit vector becomes useful.

To summarize, we assume that the mark bit vector correctly signifies the live objects on entry to the compaction algorithm. The compaction algorithm guarantees that as compaction ends and the program resumes, the alloc bit vector is properly constructed to signify the location of all allocated live objects after compaction.

The third vector is allocated specifically for use during the compaction; we call it *the offset vector*. The length of the (mark and alloc) bit vectors is proportional to the heap size and depends on object alignment. For example, in our implementation, the object alignment is eight bytes and so in order to map each bit in the bit vectors to a potential beginning of an object in memory, each bit should correspond to an 8-byte slot in memory. The overhead is 1/64 of the heap for each of these vectors. The length of the third vector depends on the platform (32 or 64-bit addressing) and on the block size, as explained in Section 2.2 below. In our implementation, we chose a block size of 256 bytes, making this vector equal in length to (each of) the other two vectors on 32-bit platforms.

2.2 The basic algorithm

In this section, we describe the basic algorithm without parallelization. In Section 3 below, we modify the basic algorithm making it run on several threads in parallel. We run the compaction in two phases. In the first phase, the **moving phase**, the actual compaction is executed, moving objects to the low end of the heap and recording some information that will later let us know where each object has moved. In the second phase, the **fix up phase**, we go over the pointers in the heap (and roots) and modify them to point to the new location of the referent object. The information recorded in the first phase, must allow the fixing of all pointers in the second phase.

The moving phase. The actual moving is simply done with two pointers. The first pointer, denoted *from*, runs consecutively on the heap (more accurately, on the mark bit vector) to find live objects to be moved. The second pointer, denoted *to*, starts at the beginning of the heap. Whenever a live object is found, it is copied to the location pointed by the *to* pointer, and the *to* pointer is incremented by the size of the copied object to point to the next word available for copying. This is done until all objects in the heap have been copied. The main issue in this phase is how to record information properly when moving an object, so that in the second phase, we can fix pointers that reference this object.

For each relocated object, we update the alloc bit vector to signify that an object exists in its new position. (The alloc bit vector is zeroed before the beginning of the compaction, so there is no need to "erase" the bits signifying previous object locations.) In addition, we divide the heap into fixed size blocks. The size of the blocks is denoted by B and in our implementation, we chose $B = 256$ bytes. For each first object in a block, we also update the offset vector to record the address to which this first object is moved. Note that the offset vector has a pointer for each block, thus, it must be long enough to hold a pointer per block. This sums up the moving phase.

The fix up phase. During the fix up phase, we go over all pointers in the heap (and also in the roots, such as threads stack and registers) and fix each pointer to reference the modified location of the referent object after the move. Given a pointer to the previous location of an object A, we can easily compute in which block the object A resides (by dividing the address by B, or shifting left eight bit positions if $B = 256$). For that block, we start by reading all mark bits of the objects in the block. These bits determine the location of the objects before they were moved. From the mark bits, we can compute the ordinal number of this object in the block. We do not elaborate on the bit manipulations required to find the ordinal number, but note that this is an efficient computation since there are not many objects in a block. For example, if a block is 256 bytes and an object is at least 16 bytes (as is the case in our system), then there exist at most eight objects in a block; therefore, our object may be first in the block, second in the block, and so forth, but at most the eighth object in the block. The number of live objects in a block is usually much smaller than eight, since many objects are not alive and since most objects are longer than 16 bytes. We actually found out that many of the searches end up outputting 1 as the ordinal number.

Returning to the algorithm, we denote the ordinal number of the referent object to be i. We now turn to the offset vector and find the location to which the first object in the block has moved. Let this offset be O. If $i = 1$, we are done and the address O is the new address of the object A. If $i > 1$, we turn to look at the alloc bit that corresponds to the heap address O. Note that this alloc bit must be set since the first object in the block has moved to address O. From that bit, we proceed to search the alloc bit vector until we find the i-th set bit in the alloc bit vector, starting at the set bit that corresponds to the address O. Because we keep the order of objects when we move them, this i-th set bit in the alloc bit vector must be the bit that corresponds to our referent A. Thus, from the alloc bit vector we can compute the new address of A and fix the given pointer.

Interestingly, this computation never touches the heap. We first check the mark bit vector, next we check the offset vector, and last, if the ordinal number is not one, we check the alloc bit vector. The fact that the fix up only touches relatively small tables, may have a good effect on the cache behavior of the compaction.

The above explanation assumes that pointers reference an object. Namely, they point to the beginning (or to the header) of the object. This assumption is correct at garbage collection times for the JVM we used. However, it is easy to modify the algorithm to handle inner-object references by searching the mark bit vector to find the object head.

A pseudo code of the fix up routine is depicted in Figure 1. It assumes a couple of routines or macros that perform simple bit operations and address translations. The *mark_bit_location* macro takes an object and returns the lo-

Procedure Fix_Up(A: Object)
begin

1. M_A= *mark_bit_location*(A)
2. M_B= *mark_bit_block_location*(A)
3. i = *ordinal*(M_A, M_B)
4. O = offset_table(M_B)
5. if (i == 1)
   - new = O
6. else
   - A_1 = alloc_bit_lcation(O)
   - A_i = location of i-th set bit in alloc_vector starting at A_1
   - new = alloc_to_heap(A_i)
7. return(new)

end

Figure 1: A pseudo code of the fix up algorithm

cation of its corresponding mark bit; the *alloc_bit_location* returns the corresponding location in the alloc bit vector. The *alloc_to_heap* macro receives a location in the alloc bit vector and returns the heap location of the object that corresponds to this location. The *mark_bit_block_location* macro returns the location in the mark bit vector that corresponds to the beginning of the block that contains the object A. The routine *ordinal* receives two locations in the mark bit vector and counts the number of ones between them. In particular, it returns the ordinal number of the object corresponding to the first parameter in the block corresponding to the second parameter. The offset_table of a given block location returns the location to which the first object in this block has been moved.

A possible optimization, that we adopt in our JVM, is applicable when the minimal size of objects, denoted S, exceeds the alignment restriction, denoted A. For example, in our system, objects are 8-bytes aligned and their minimal size is 16, i.e., $A = 8$ and $S = 16$. It is then possible to keep a bit per S bytes rather than per A bytes, as described above. Information kept in this manner allows finding the ordinal number, which is enough for the mark bit vector, but does not allow computing the address of an object from the location of its corresponding bit; therefore, this cannot be done with the alloc bit vector. Using this observation, we may save $1 - A/S$ (half, in our setting) of the space used by the mark bits. Our measurements show that performance is not harmed by employing this space saving.

3. MAKING THE COMPACTION PARALLEL

We start by splitting the heap into areas so the compaction work can be split between threads. As always, there is a tradeoff in determining the number of areas. For good load balancing (and therefore good scalability) it is better to use a large number of small areas. But for low synchronization and low management overhead, it is better to have a small number of large areas. Synchronization does not play a substantial role in the tradeoff, since synchronizing to get the next area does not create a noticeable delay in execution time. However, the quality of compaction and the overhead of managing these areas are significant. A typical choice in our collector is to let the number of areas be 16 times the number of processors, but the area size is limited to be at least 4 MB.

An important new idea in our parallel version of the compaction is to allow objects to move from one area to another. In the beginning, a compacting thread pushes objects to the lower addresses of the area it is working in. However, when some areas have been compacted, the threads start to compact other areas into the remaining space within the areas that have already been compacted. This method is different from previous schemes [7] that partitioned the heap into a small number of large areas and compacted each area separately. In those schemes, the heap is compacted into several dense piles of objects, depending on the number of areas. Therefore, a good quality of compaction dictates a small number of areas, which may be bad for load balancing. Using our idea, the entire heap is compacted to the low addresses, except for some negligible holes remaining in the last areas, which were not completely filled. Using this idea, we get the best of both worlds: better compaction quality as objects are all moved to the lower end of the heap and better scalability as small areas allow better load balancing.

Getting more into the details, we keep an index (or a pointer) that signifies the next area to be compacted. A thread A obtains a task by using a compare and swap synchronization to increment this index. If successful, the thread A has obtained an area to compact. If the compare and swap fails, another thread has obtained this area and A tries the next area. After obtaining an area to compact, A also looks for a target area into which it can output the compacted area; it only tries to compact into areas with lower addresses. A compaction is never made to an area with higher addresses. This invariant may be kept, because an area can always be compacted into itself. We keep a table, denoted the *target table* of areas, that provides a pointer to the beginning of the free space for each area. Initially, the target table contains nulls, implying that no area may be compacted into another area. However, whenever a thread compacts an area into itself (or similarly into another area), it updates the values of the compacted area entry (or the target area entry, if different) in the target table to point to the beginning of the remaining free space in the area. Returning to a thread A that is looking for a target area, once A has found a lower area with free space, it tries to lock the target area by using a compare and swap instruction to null the entry of the area in the target table. If this operation succeeds, A can compact into the selected target area. No thread will try to use this area as a target, as long as it has a null entry. After finishing the compaction, A updates the free space pointer of the target area in the target table. At that time, this area may be used as a target area by other threads. On the other hand, if the target area gets filled during the compaction, its remaining space entry will remain null and thread A will use the above mechanism to find another target area to compact into.

Procedure Parallel_Move()
begin

1. S= *fetchAndAdd*(*lastCompacted*)
2. If ($S == N$) exit.
3. for ($T = 0$; $T < S$; $T++$)
   - *pCopy*= *fetchAndSet*($P(T)$,NULL)
   - if (pCopy != NULL) break
4. if ($T == S$)
   - Compact S internally
   - $P(S)$= resulting free space pointer
   - goto step 1
5. else Compact remaining S objects into T using *pCopy*
   - If (T becomes full) goto step 3
   - If (S becomes empty)
     - P(S) = S
     - P(T) = (updated) pCopy
     - goto step 1

end

Figure 2: A per-thread pseudo code of the parallel move algorithm

Normally, the size of a block is much smaller than the size of an area. (In our implementation it was 256 bytes and 4MB respectively.) When moving a block, we never split its objects to more than one area. In other words, if the remaining free space in the end of an area does not suffice to move all live objects that start in a block, then we abandon the area and start moving blocks to a new area. Thus, the live objects of a block are always moved consecutively.

Figure 2 illustrates our parallel compaction algorithm. Suppose the heap is divided to N areas (0 .. N-1), and for each area A we keep a pointer (denoted $P(A)$) to its free space (initially set to NULL). We also maintain a global variable with the index of the last area that is compacted (denoted *lastCompacted*). We define the atomic operations of *fetchAndAdd*(V) (which increments V and returns its old value) and *fetchAndSet*(V,$newValue$) (which sets V to $newValue$ and returns its old value) [1].

4. TRADING QUALITY OF COMPACTION FOR HIGHER EFFICIENCY

In our algorithm (presented in Sections 2 and 3 above), the heap is (almost) perfectly compacted into the low addresses of the heap so that the compacted objects are dense with no waste of space. We now suggest trading this quality of compaction for reduced overhead both in execution time as well as in the amount of auxiliary space used.

In our main algorithms, we use blocks of length B bytes and we spend a considerable effort compacting the objects within a block. In terms of space overhead, this intra-block compaction necessitates the use of the mark bit vector. In terms of runtime overhead, the full compaction requires updating and using the mark bit vector and the alloc bit vector for the pointer fix up. The main idea behind the relaxed compaction algorithm that we present now, is to perform compaction between blocks, but not within blocks. Thus, for each block σ we keep only an offset $\Delta(\sigma)$ signifying that all objects within the block σ are moved $\Delta(\sigma)$ words backwards. The value of $\Delta(\sigma)$ is determined so that the first live object in the block moves to the first available word after the previous blocks have been compacted. In particular, $\Delta(\sigma + 1)$ is set so that the first live object in Block $\sigma + 1$ moves just after the last live object in block σ ends. If a block has no live objects, it consumes no space in the compacted heap. We simply let the next non-empty block move its first live object to the next available word. Figure 3 depicts a compaction of two possible first blocks in a heap.

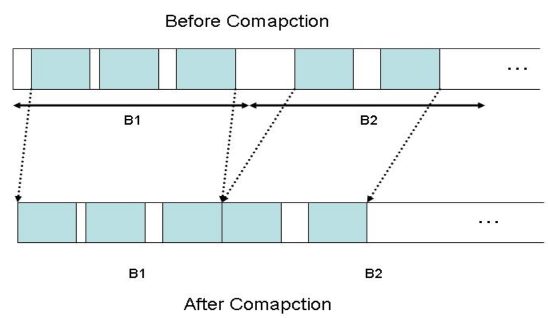

Figure 3: Inter-block Compaction: free spaces within the block are not compacted

Clearly, if a block has only one live object or no live objects, it is compacted perfectly, leaving no wasted space. Also, if a block is full of objects with no space between them, it is also perfectly compacted. Thus, dense live objects and sparse live objects may yield good compaction quality. However, in-between these two cases, we lose space. The worst placement of objects in a block for this method is the placement of two small objects, one residing in the beginning of the block and the other in the end of the block.

We have implemented this optimization and measured its behavior. Interestingly, we found that with the SPECjbb2000 benchmark and heap size inducing 60% heap consumption[2], the cost of this optimization is as low as 3% increment in the size of the compacted area, the reduced compaction quality did not change the overall throughput, compaction time was reduced by approximately 20%, and the space overhead (for compaction data structures) was reduced by 66%. These measurements are presented in Section 7.3.

[1]Both operations may be constructed using the compare and swap atomic operation, which is given a variable, its expected old value, and a required new value. It succeeds to set the variable to the new value only if the old value was found. In the case of a few threads trying to perform this operation simultaneously, only one will return as successful.

[2]Namely, the accumulated size of the live objects is 60% of the heap size.

5. THE ORDER OF OBJECTS MATTERS

The two-finger compaction algorithm was designed for compacting LISP heaps containing objects of one size only. The algorithm may be visualized via two fingers (i.e., pointers). One finger traverses the heap from the beginning upwards searching for an empty location, while the other finger moves from the end of the heap to the beginning looking for a live object. When the two fingers stop their move, the lower finger points to a hole and the upper finger points to an object. Since all objects are of the same size, all holes are also of the same size and we can move the object of the upper finger to the hole found by the lower finger. Then, the two fingers go on searching. When the two fingers meet each other, the compaction is done. Forwarding pointers are left in the moved objects so that pointers to moved objects can later be updated. (For more details see [8]). The big advantage of this method is that it typically moves only half of the objects, whereas other methods move all objects. This also implies less work on fixing up the pointers, since many pointers (referencing addresses below the point where the two fingers meet) do not need to be updated. Also, this algorithm can be easily made parallel; the fix up is simple and efficient, and no space overhead is required.

The disadvantage of this compaction method is that it does not preserve the order of live objects. According to folklore belief, this does not yield a good compaction. First, it is believed that locality of reference holds with respect to allocation order; thus, the paging and caching behavior of the program deteriorates when the order of objects is not preserved by the compaction. Second, objects allocated together tend to die together and thus, not keeping objects with their allocated neighbors may increase future fragmentation. In spite of this folklore belief, we made an attempt to implement an adaptation of the two-finger method for Java and to check its behavior. An adaptation is required since the objects in a Java heap differ in size. We also tried to design an adaptation that can later run in parallel threads. The hope was that the negative effects would be small on a modern system and that the advantages of the algorithm would win. As it turned out, the throughput of the program using this compaction algorithm deteriorated and the compacted heap obtained by this algorithm became fragmented by subsequent program allocations and garbage collections. This validates the folklore belief.

The modified algorithm is now described briefly. In this description, we do not care about the efficiency of the compaction itself. As the compaction quality is the focus of this discussion, we simplify the presentation by describing a naive and inefficient algorithm. We will later measure the throughput of the application without taking into account the running time of the compaction[3]. Also, the quality of the compaction does not depend on the running time of the compaction itself.

Getting into the algorithmic details, Java objects are partitioned into *small* and *big* objects. The threshold size is a tunable parameter, and a typical threshold setting is 1 kilobyte. The algorithm starts by finding the lowest address ℓ in the heap that ensures that all small objects in addresses above ℓ can be moved to holes below ℓ. Naively looking for ℓ may require a lot of computation but, as stated above, in this short description we ignore all algorithmic efficiency issues.

After finding the address ℓ, we move small objects from addresses above ℓ to addresses below ℓ. We start by moving objects to holes of exactly the same size, where both holes (starting from the beginning of the heap) and objects (starting from ℓ) are processed by their order in the heap. After fitting exactly matching objects and holes, we try to fit smaller objects into bigger holes; still, all objects and holes are small. Next, we put small objects in large holes. Finally, if large holes remain, we also move large objects into large holes, wherever possible. Moving large objects to large holes is done using the following strategy: go over the holes according to their sizes in ascending order; for each hole, go over objects smaller than the hole according to their size in descending order and move the objects into the hole. For each object moved to a hole, we leave a forwarding pointer.

At this stage, we are done with moving objects to holes, but some large objects may still be scattered above address ℓ and need to be moved. These objects are compacted by sliding them down to the address ℓ. This compaction operation requires a small auxiliary table for a later fix up of the heap pointers. Before starting the slide, we run an update on all pointers in the heap. This fix up process starts by computing the final address for each large object that has not yet been moved. This address is written into the small auxiliary table. Next, the fix up process goes over all pointers in the heap and updates pointers according to the forwarding pointer in the object or the information stored in the auxiliary table. Finally, the remaining large live objects above address ℓ are compacted towards the address ℓ and we are done.

In Section 7.4 below, we provide measurements of this algorithm. It turns out that fragmentation quickly recurs after several garbage collections and that program throughput is reduced with this compaction algorithm. This does not happen when using the original compaction algorithm or the compaction algorithm described in this paper (in Section 2).

6. IMPLEMENTATIONS

We have implemented our algorithms on top of the mark-sweep-compact collector that is part of the IBM production JVM 1.4.0. This collector is described in [5, 6]. The compact part of this collector implements the threaded algorithm of Jonkers [9] and Morris [12]. We have replaced this compactor with our parallel compaction, which is run on n threads in parallel, when using an n-way machine. In order to compare between our algorithm and the original threaded algorithm, we sometimes restricted our algorithm to use a single thread.

Non-movable objects. Our platform is not type accurate. Although each field in the heap can be correctly identified as a pointer or non-pointer, we cannot make the distinction between pointer and non-pointer fields in the stack and

[3]This is the default when measuring the SPECjbb2000 benchmark. Compaction happens during ramp-up while changing the number of warehouses. That time is not accounted for when computing the throughput.

registers (the roots). Thus, some objects cannot be moved, and are left in their original location during the compaction. This complicates any compaction algorithm and specifically, forced us to deal with (normally irrelevant) technicalities. We do not get into the details in this paper. We just note that the number of such objects is small (around 1% by our measurements) and thus we do not expect to see them influencing the compaction behavior. Furthermore, we compare our algorithm to a base algorithm that deals with the same problem, in a similar manner, and thus, there are no hidden advantages favoring our algorithm in the measured comparison.

7. MEASUREMENTS

In this section, we present the results of implementing our parallel compaction algorithm into the IBM production JVM. We start by describing our test environment. We then compare the measurements of the new compaction algorithm to those of the threaded algorithm previously used with the IBM production JVM. The threaded compaction algorithm was presented in [9, 12] and is also described in [8]. Next, we present measurements for our inter-block compaction that trades compaction quality with time and space overhead (as described in Section 4 above). Finally, we present the measurements for the extended two-finger compaction algorithm that does not preserve the order of compacted objects. (This algorithm was described in Section 5 above.)

7.1 Platform and benchmarks

Our measurements were taken primarily on a pSeries server, with eight 600 MHz PowerPC RS64 III processors (64-bit) and 5 GB of RAM running AIX 5.1.

Measurements of the extended two-finger compaction algorithm were done on an IBM Netfinity 7000 server, with four 550 MHz Pentium III XeonTM processors (32-bit) and 2 GB of RAM, running Windows 2000. Measurements of our compaction on a Web server application were done on an IBM xSeries 440 Server with four 2.0 GHz Intel XeonTM MP processors (32-bit, with Hyper-Threading enabled) and 3.5 GB of RAM, running Windows 2000.

Benchmarks. We used three benchmarks: *SPECjbb2000*, *Trade3*, and the *SPECjvm98* benchmark suite.

SPECjbb2000 [14] is a Java business benchmark inspired by TPC-C. It emulates a 3-tier system, concentrating on the middle tier. SPECjbb is throughput oriented: it measures the amount of work (i.e., number of "transactions") done during a given time. The result is given in *TPM* (transactions per minute). On an 8-way multiprocessor, an official run of SPECjbb includes sixteen short cycles of two minutes each. The first cycle creates a single warehouse, and each successive cycle increases the number of warehouses by one. Adding warehouses increases the amount of live objects. SPECjbb issues a *TPM* score for each cycle, and a total score for the entire run.

SPECjbb2000 initiates a forced GC (or System.gc() call [1]) before each cycle. With the IBM JVM 1.4.0, compaction occurs if a System.GC has been requested and the last GC did not compact [6]. So IBM JVM 1.4.0 initiates a compaction before each SPECjbb cycle. As each cycle increases the heap residency, this gives us an excellent opportunity to test compaction in different heap conditions. On the other hand, throughput measurements are paused when these compactions occur, so compaction time does not influence the SPECjbb2000 scores. Therefore, we provide measurements of the actual compaction times as well as the throughput.

Another measurement we ran was one that executes a compaction once every 10 or 20 garbage collections. That gave us an indication of the effect of compaction on the throughput when both its benefit to the order of the heap and its cost in time affects the throughput scores.

Trade3 [16] is the third generation of the WebSphere [13] end-to-end benchmark and performance sample application. This benchmark models an online stock brokerage application and uses a real world workload, driving WebSphere's implementation of J2EE 1.3, Web Services, and other key Web application server components. Trade3 execution involves three components: a client application which generate the workload, a database, and a Web Application Server. The database and the Web Application Server were installed on separate machines. The Web Performance Tool (WPT) was used to generate the workload. WPT is an HTTP engine capable of simulating hundreds of HTTP clients. We used its default test driver (AKstress) and test script, but instead of simulating a single client issuing 2000 HTTP requests, we have simulated 300 clients and let it run for 30 minutes. We have measured the performance of the main component (the Web Application Server), and relate to the JVM of this application in the rest of this section. The performance metrics we display is requests per seconds.

Trade3 is interesting for compaction research, as it creates a significant amount of fragmentation in the Java heap.

SPECjvm98 [15] is a benchmark suite that measures computer system performance for JVM client platforms. It consists mostly of single-threaded benchmarks that use relatively small heaps (typically, less than 50 MB). We measured this benchmark suite in order to provide some insight into the behavior of our compaction algorithm for small applications. All benchmarks in this suite initiate a forced GC (and hence a compaction) before and after their measured period. One of the benchmarks (javac) also initiates a few compactions inside the measured period.

Test rules. All measurements presented in this section are results averaged from at least five runs, except for fragmentation results, which were computed from one typical run. In order to test the compaction mechanism under a reasonably heavy load when running SPECjbb, we aimed at achieving a 60% heap residency at sixteen warehouses, and therefore used a 600 MB heap. For the same reason, Trade3 was run with a 180 MB heap. The entire SPECjvm98 suite was run with a 32 MB heap.

7.2 The main parallel algorithm

We start with the measurements of the main parallel algorithm. We measure its efficiency versus the original threaded algorithm that was used with the IBM production JVM before this algorithm was incorporated.

When evaluating a compaction algorithm, the interesting parameters are: **compaction time** (i.e., the time it takes to compact the entire heap), **program throughput**, **compaction quality**, and (for a parallel compaction algorithm) the **speedup** of the compaction on a multiprocessor.

In this section, we present detailed comparisons of the original collector (denoted the *threaded* compaction) with a single-threaded version of our new compaction (denoted *parallel-restricted* compaction), and with our parallel compaction (denoted *parallel* compaction), when running SPECjbb2000.

Compaction times. Starting with compaction time, the running times of our *parallel-restricted* algorithm are compared with the running times of the *threaded* compaction algorithm. Figure 4 shows the compaction times of each of the algorithms as a function of the number of warehouses, which implies the heap residency of the compacted heap. It is easy to see that the larger the size of the live objects, the more saving we get when running the new compaction algorithm. We stress that this is done without parallelism. Thus, the superiority of our method is fairly measured against the original non-parallel threaded compaction algorithm. When using parallelism, our method outperforms the original compacting algorithm by far. The advantage of the parallel algorithm (even when executed by a single thread) is presented by the reduction in compaction time in Figure 5.

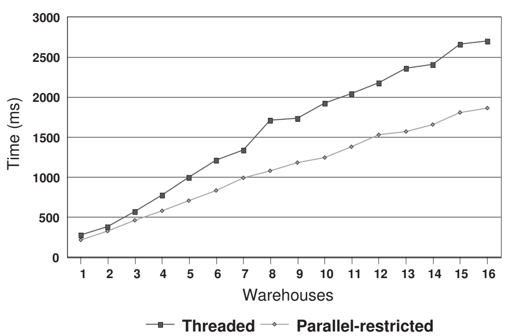

Figure 4: SPECjbb: Compaction time comparison between the threaded compaction and our new compaction (single-threaded implementation)

Figure 5: SPECjbb: Compaction time reduction with our new compaction (single-threaded implementation) relative to the threaded compaction

The reported measure is the average compaction time. We do not report the maximum times, as the results are quite stable. In particular, for SPECjbb, the maximal compact time with the new parallel compaction algorithm never exceeds 6% above the average time. On the other hand, with the original threaded compactor the maximum compaction time never exceeded an increase of 12% above the average.

Throughput. As mentioned above, the SPECjbb2000 does not take compaction time into account in its throughput measurements. Thus, an extremely long compaction would not make a difference in the throughput measurements. The throughput measurements only measure the effect of the compaction on program efficiency. As a sanity check, we measured the throughput with the new parallel compaction algorithm and compared it with the throughput of the JVM when using the threaded compaction algorithm. The results indeed show no noticeable difference and are presented in Figure 6.

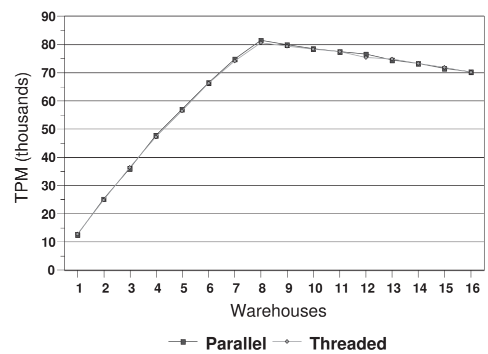

Figure 6: SPECjbb: Throughput comparison between the threaded compaction and the parallel compaction

Speedup. Finally, we check the speedup of the parallel compaction when varying the number of threads running the compaction. We used an 8-way multiprocessor and ran the compaction on two, four, six or eight threads. The speedup computed is the compaction running time with a single thread divided by the compaction time with several threads. The results are presented in Figure 7. It turns out that the scalability of this algorithm is high, yielding speedup factor of 2.0 on two threads, 3.7 on four threads, 5.2 on six threads, and 6.6 on eight threads.

Compaction's effect on throughput. Running compaction poses a trade-off to the JVM implementer. On one hand, the compaction improves the heap order, reduces fragmentation and increases allocation efficiency. On the other hand, compaction takes time. We measured the throughput of the SPECjbb benchmark when additional compaction executions were forced on top of the original SPECjbb invocations of compaction. We ran this measurement twice. Once, forcing compactions every 10 collections, and again, forcing compactions every 20 collections. This test was run with the threaded compaction, with the restricted version of our

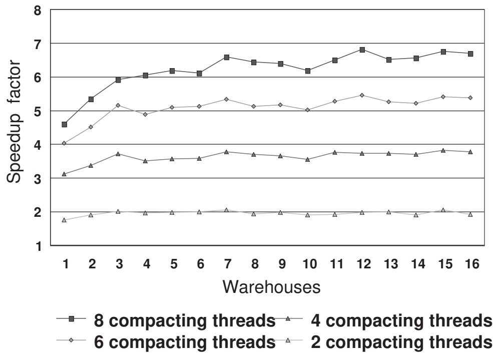

Figure 7: SPECjbb: Speedup comparison when running the parallel compaction with different number of threads

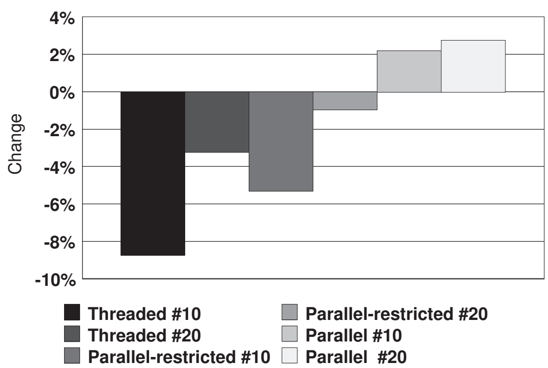

Figure 8: SPECjbb: Throughput comparison. Forcing compaction every 10 and 20 collections with the threaded compaction, the parallel-restricted (single threaded) compaction, and the parallel compaction (8 threads). All scores are compared with a normal run of threaded compaction

parallel compaction that runs on only a single thread , and finally, with the parallel compaction (on eight threads). Figure 8 depicts the throughput change relative to the regular execution (without additional compactions) of the threaded compaction. #10 denotes forcing a compaction every 10 collections and #20 every 20 collections. We can see that the threaded compaction is too slow to allow additional compactions to yield an overall throughput improvement. Actually, running the original threaded algorithm reduces the overall throughput by more than 8%. Only the parallel compaction is fast enough to improve the overall time when run every 10 or 20 collections.

The Trade3 benchmark. Fragmentation reduction is one of the major motivations for compaction [8]. In the SPECjbb benchmark, fragmentation does not appear to be significant, although forcing compaction frequently may still improve the throughput. The Trade3 benchmark creates more fragmentation in the Java heap and allocates large objects that require large consecutive heap spaces. Thus, the built-in (conservative) triggering mechanism triggers compactions with this benchmark. In our measurements of runs of Trade3, the original IBM JVM 1.4.0 triggered compaction about every 90 GC cycles. In addition, we forced compaction every 20 GC cycles. The average increase in free space due to compaction was around 8% of the size of the heap, for both the threaded and parallel compaction algorithms. Without forcing compaction every 20 GC cycles, the fewer compactions resulted in a higher increase in free space; around 13% of the size of the heap, for both the threaded and the parallel compaction algorithms

Compaction Type	Compact time (ms)		Improvement over Threaded	
	default trigger	#20 trigger	default trigger	#20 trigger
Threaded	1698	1671	–	–
Parallel-restricted (single threaded)	1387	1251	18.3%	25.1%
Parallel	499	440	70.6%	64.8 %

Table 1: Trade3: Compact times for the threaded compaction, the Parallel-restricted, and the parallel compaction; with and without forcing compaction every 20 GC cycles

Compaction Type	Requests per second		Improvement over Threaded	
	default trigger	#20 trigger	default trigger	#20 trigger
Threaded	219.8	224.5	–	–
Parallel-restricted (single threaded)	221.7	226.1	0.9 %	0.7%
Parallel	222.4	229.1	1.2 %	2.0%

Table 2: Trade3: Throughput for the threaded compaction, the Parallel-restricted, and the parallel compaction; with and without forcing compaction every 20 GC cycles

In what follows, we present measurements results for Trade3. In Tables 1-2 the threaded algorithm is compared with the restricted (single threaded) version of the parallel algorithm and with the (unrestricted) parallel algorithm running in parallel on all the processors on the 4-way xSeries 440 Server. Two measurements are reported. First, the compaction times, when the algorithms are run on the Java heap as induced by the Trade3 benchmark. This relates to application's pause times. Measurements were taken both on the original built-in compaction triggering mechanism (denoted *default trigger*), which executed compaction about once every 90 GC cycles, and also when forcing additional compactions every 20 collections (denoted *#20 trigger*). Second, we report the overall throughput of the application when using each of these compaction methods.

In contrast to SPECjbb, in which compaction effort is not accounted for in the throughput measure, we can see a throughput increase when the more efficient compaction is used. The compaction time is reduced even when using our algorithm on a single thread. When it is run on all four processors, the speedup obtained is more than 2.8, which is smaller than the speedup observed on the PowerPC platform. As with SPECjbb, the compaction times did not vary too much. The maximum compaction time exceeded the average compaction time by at most 22% with the parallel compactor and by at most 14% with the threaded compactor. Thus, the new collector is still faster even on a single thread and even when considering the maximum compaction time.

Behavior with client applications. As a sanity check, we ran our compaction algorithm with the SPECjvm98 benchmark suite on a uniprocessor. In these benchmarks, there is a minor need for compaction and usually no need for parallelism. Nevertheless, our new algorithm behaves well even in this setting. We compared the compaction running times to those of the threaded compaction. The results, which are presented in Table 3, reflect the average running time of a compaction run towards the end of the run of the benchmark. On most benchmarks, the parallel compaction running on a uniprocessor does better than the threaded compaction.

	Compact time (ms)		Total time (sec)	
Benchmarks	Threaded	Parallel	Threaded	Parallel
compress	28.3	21.1	16.7	16.6
jess	19.2	19.1	12.0	11.9
db	14.6	13.9	22.4	22.3
javac	21.2	24.4	24.5	24.4
mtrt	16.0	16.3	7.8	7.9
jack	80.8	77.8	15.1	14.0
raytrace	15.7	15.9	7.5	7.5

Table 3: SPECjvm98: Compact times and total times for both the threaded compaction and the parallel compaction

We also checked the influence of the compaction on the program throughput. As the heaps are relatively small, compaction did not significantly influence the overall times. Thus, both JVMs gave similar throughput results.

Auxiliary space usage. Our algorithm uses three auxiliary vectors, which require 2.3% or 3.9% additional space over the heap. To check whether this extra space buys us any of the improvements shown earlier, we tried to provide the threaded algorithm with 4% more heap space and check the improvement in throughput. The results are presented in Table 4. We did not include a normal run of SPECjbb since compaction time is artificially not included in the measurement. SPECjbb #10 refers to a run of SPECjbb in which we forced additional compactions every 10 collections. SPECjbb #20 refers to forcing additional compactions every 20 collections. Trade3 #20 refers to a run of Trade3 in which we force additional compactions every 20 collections. As in all other runs, SPECjbb was run on a heap size of 600MB and Trade3 was run on a 180MB heap.

It turns out that adding more space to the heap does not yield the benefits of a good algorithm that uses additional auxiliary space. Also, note that the main advantage of a better compaction is the reduction in the pause time, as compaction creates long pauses. In this respect, using a larger heap only increases the pause time, whereas the new parallel algorithm reduces the pause times substantially (see Figures 4, 5, and 7).

Benchmarks	Improve with 4% add. space (threaded alg)	Improve with restricted-parallel (no add. space)
SPECjbb #10	1.46%	3.76%
SPECjbb #20	0.72%	2.33%
Trade3 #20	-0.02%	0.73%

Table 4: Throughput improvement when adding 4% extra heap space to the basic collector (employing the threaded compaction), and throughput improvement when replacing the threaded compaction with the new parallel algorithm restricted to running with a single thread.

7.3 Trading quality for efficiency

Next, we move to checking the inter-block compaction variant proposed in Section 4. This variant is denoted *inter-block* in the figures. We start by measuring the compaction times for this method. We expect to see lower running times and indeed this is the case, as shown in Figure 9. The improvement in running times is presented in Figure 10 below. As one can see, the improvement is around 25%. All these measurements are done while running the compaction in parallel, in its typical form.

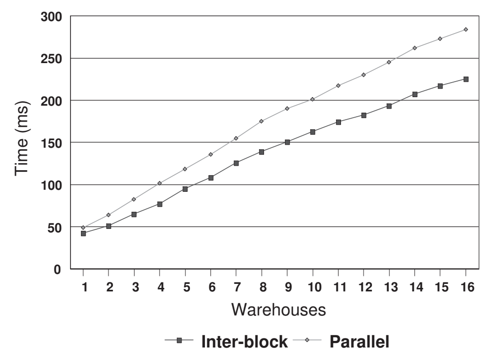

Figure 9: SPECjbb: Compaction time comparison between the parallel compaction and its inter-block variant

It is interesting to check, in this case, how the throughput is influenced by the change. Because the compaction quality is reduced, this may affect the throughput of the program. Our SPECjbb2000 measurements show no throughput change between the parallel compaction and its inter-block variant; differences, in all warehouses, are smaller than 0.25%.

Finally, it is interesting to measure how much the compaction quality was reduced by the inter-block variant. For each number of warehouses, we check the size of the compacted area of the live objects just after compaction. For

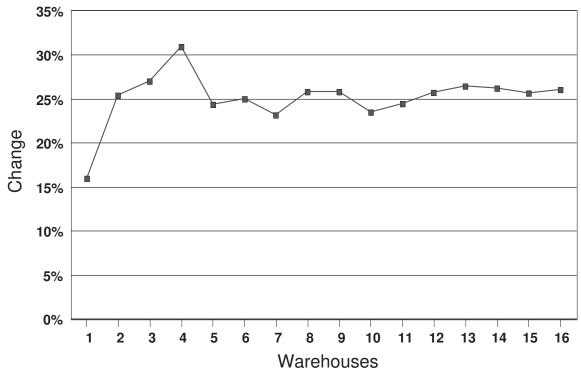

Figure 10: SPECjbb: Compaction time reduction with the inter-block variant relatively to the parallel compaction

the inter-block variant, this should be higher since some of the compaction was not performed. The increase in the size of the compacted area is presented in Table 5.

Warehouses	W 4	W 8	W 12	W 16
Compacted area size (MB): parallel	107.7	199.7	276.5	351.5
Compacted area size (MB): inter-block	111.8	205.0	282.6	358.3
Compacted area size increase	3.8%	2.7%	2.2%	1.9%

Table 5: SPECjbb: Size (and increase in size) of the compacted areas for the parallel compaction and its inter-block variant, for selected warehouses

7.4 The extended two-finger algorithm

We now turn to the extended two-finger algorithm, denoted *extended-two-finger* in the figures. The extended two fingers algorithm does not preserve the order of objects. The most interesting measure is how this influences the throughput. As compaction time is not accounted for, the throughput differences do not reflect the time it takes to run a compaction, but only the effect of ordering objects arbitrarily. The results, compared to the original threaded algorithm are presented in Figure 11 below.

It is easy to see that as the heap residency grows, the throughput is dramatically reduced.

We have also measured the distribution of the lengths of free chunks, to try and account for the throughput difference by fragmentation (i.e., the fact that free chunks become smaller on average). We present these measurements in Table 6.

In this table, we report the percentile of free space that resides in all chunks that are smaller than a given chunk size. We do that for a few chunk sizes. For example, the row labeled *64 bytes* shows the percentile of free space that resides in all chunks smaller than 64 bytes. We compare the threaded compaction with the extended-two-finger compaction. Measurements were taken immediately after the

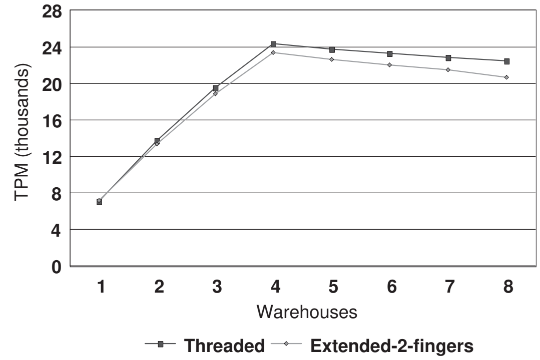

Figure 11: SPECjbb on a 4-way: Throughput comparison between the threaded compaction and the extended-two-finger compaction

Size limit	After Compact		After Compact + 10 GCs	
	Threaded	2-fingers	Threaded	2-fingers
32 bytes	0.0%	0.2%	1.2%	1.4%
64 bytes	0.0%	0.3%	2.7%	5.3%
128 bytes	0.0%	0.3%	3.2%	8.6%
256 bytes	0.0%	0.3%	3.8%	9.9%

Table 6: SPECJbb on a 4-way: the percentile of free space in all chunks below given sizes, after compaction and ten GCs later, for the threaded compaction and the extended-two-finger compaction

compaction that preceded the cycle with eight warehouses, and ten GCs later.

The table shows that immediately after the compaction, the distribution of small chunks is similar in both compaction methods. Yet, when using the extended-two-finger method, the amount of small chunks increases much faster than when using the threaded method. We believe that this occurred because jointly allocated objects are also freed together. As the extended-two-finger takes blocks of such objects and sprinkles them all over the heap, the cured fragmentation reappears as these objects are freed. From the table it is clear that this phenomena happens, yet, it does not seem strong enough to justify all the reduced throughput. We thus tend to believe that the main reason for the reduced performance is the reduced locality of reference during the program run.

We remark, without presenting the actual table, that a similar effect is not seen with the inter-block method. There, as the order of objects is maintained, the throughput is steady and the distribution of free chunks looks similar to that of the parallel compaction.

8. CONCLUSIONS

We have presented a parallel compaction algorithm adequate both for server SMPs as well as for client uniprocessors. Our algorithm is highly efficient; it scales well with good load balancing and provides a good quality of compaction, as all objects are being pushed to the bottom of the heap, rather than being collected in several piles to allow parallelism. This algorithm has been implemented and measured on the IBM production JVM and has been incorporated into the production JVM due to its qualities.

We further suggested a trade-off between compaction quality and overhead, demonstrating how a somewhat reduced compaction quality can substantially improve performance.

Finally, we have revalidated traditional wisdom, by checking a variant of the two-finger algorithm on the Java platform. The two-finger algorithm does not preserve object ordering. We have confirmed the wide belief that such compaction hinders program efficiency and eventually creates more fragmentation.

9. ACKNOWLEDGMENTS

We thank Ori Ben Yitzhak, Elliot (Hillel) Kolodner, Victor Leikehman (from the IBM Research Laboratory in Haifa), and Avi Owshanko (from the Dept. of Computer Science, Technion, Haifa) for useful discussions. We are particularly grateful to Sam Borman, Martin Trotter, and Robert Berry from the IBM Java Technology Center in Hursley, for their collaboration and support. We are also grateful to Rob Geiner from the IBM WebSphere Performance group in Raleigh and Nikola Grcevski from the IBM Java JIT Performance in Toronto for their generous help in enabling some of the measurements included in this paper.

10. REFERENCES

[1] java.lang class system. http://java.sun.com/j2se/1.3/docs/api/java/lang/System.html.

[2] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[3] David F. Bacon, Stephen J. Fink, and David Grove. Space- and time-efficient implementation of the java object model. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 111–132. Springer-Verlag, 2002.

[4] 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.

[5] Sam Borman. Sensible Sanitation - Understanding the IBM Java Garbage Collector (Part 1: Object allocation). http://www.ibm.com/developerworks/ibm/library/i-garbage1.

[6] Sam Borman. Sensible Sanitation - Understanding the IBM Java Garbage Collector (Part 2: Garbage collection). http://www.ibm.com/developerworks/ibm/library/i-garbage2.

[7] 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.

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

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

[10] 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.

[11] 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.

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

[13] WebSphere Software platform. Ibm software products. http://www.ibm.com/software/info1/websphere/index.jsp.

[14] SPECjbb2000 Java Business Benchmark. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jbb2000/.

[15] SPECjvm98 Benchmarks. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jvm98/.

[16] Trade3 Web Application Server Benchmark. IBM Software. Available at http://www.ibm.com/software/webservers/appserv/-benchmark3.html.