Heap Compression for Memory-Constrained Java Environments [*]

G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin
CSE Department
The Pennsylvania State University
University Park, PA 16802
{gchen,kandemir,vijay,mji}cse.psu.edu

B. Mathiske
Sun Microsystems, Inc.
4220 Network Circle
Santa Clara, CA 95054
Bernd.Mathiske@sun.com

M. Wolczko
Sun Microsystems, Inc.
2600 Casey Ave.
Mountain View, CA 94043
mario@eng.sun.com

ABSTRACT

Java is becoming the main software platform for consumer and embedded devices such as mobile phones, PDAs, TV set-top boxes, and in-vehicle systems. Since many of these systems are memory constrained, it is extremely important to keep the memory footprint of Java applications under control.

The goal of this work is to enable the execution of Java applications using a smaller heap footprint than that possible using current embedded JVMs. We propose a set of memory management strategies to reduce heap footprint of embedded Java applications that execute under severe memory constraints. Our first contribution is a new garbage collector, referred to as the Mark-Compact-Compress (MCC) collector, that allows an application to run with a heap smaller than its footprint. An important characteristic of this collector is that it compresses objects when heap compaction is not sufficient for creating space for the current allocation request. In addition to employing compression, we also consider a heap management strategy and associated garbage collector, called MCL (Mark-Compact-Lazy Allocate), based on lazy allocation of object portions. This new collector operates like the conventional Mark-Compact (MC) collector, but takes advantage of the observation that many Java applications create large objects, of which only a small portion is actually used. In addition, we also combine MCC and MCL, and present MCCL (Mark-Compact-Compress-Lazy Allocate), which outperforms both MCC and MCL.

We have implemented these collectors using KVM, and performed extensive experiments using a set of ten embedded Java applications. We have found our new garbage collection strategies to be useful in two main aspects. First, they reduce the minimum heap size necessary to execute an application without out-of-memory exception. Second, our strategies reduce the heap occupancy. That is, at a given time, they reduce the heap memory requirement of the application being executed. We have also conducted experiments with a more aggressive object compression strategy and discussed its main advantages.

[*]This work was supported in part by NSF grants 0073419, 0103583, CAREER 0093082, CARREER 0093085

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*

General Terms

Algorithms, Languages

Keywords

Java Virtual Machine, garbage collection, heap, memory compression

1. INTRODUCTION

The market for mobile devices and phones is continuing to increase at a rapid rate. For example, the handheld mobile device market in the U.S. is currently increasing at an annual rate of 22% [9]. In contrast to the PC market, several products and companies vie for this market share. Since the ability to support dynamic software content is a major factor in determining the success of mobile devices, many of the mobile device manufacturers are increasingly adopting Java technology. A recent report projects that Java will be the dominant terminal platform in the wireless sector, being supported by over 450 million handsets in 2007, corresponding to 74% of all wireless phones that will ship that year [31].

Using the Java technology on personal information devices has several important benefits [5]. First, Java is cross-platform compatible. As a result, Java code can run smoothly without modification on a wide range of devices. Such cross-platform compatibility is especially important for the diverse mobile device market that is shaped by a variety of devices executing different operating systems. Second, Java enhances user experience by supporting rich GUI components and by providing dynamic downloading capability. Further, Java security model allows users to run these applications on their devices safely. Finally, Java has a very mature developer community. The developer talent needed for Java devices already exists and is readily available.

Memory Problem. However, there exist several challenges in supporting Java applications in mobile devices. For example, the memory space and energy supply of mobile devices impose an entirely different set of constraints as compared to high-end Java based environments. Many mobile devices have stringent memory requirements as memory has severe implications on the cost, form factor, and energy consumption of the device. The budget for the memory has a significant impact on the overall cost of a device. For example, currently, a Palm m130 PDA with an 8MB memory costs $199 as compared to $299 cost for a similar configuration with a 16MB memory [7]. Similarly, as compared to the base price of $99 for a Palm Zire PDA with 2MB memory, upgrading its memory to 8MB adds an additional $59 [1]. In addition to the cost factor, a larger memory also demands a larger form factor. In fact, many cell phone companies need to resort to more costly packaging techniques to incorporate larger memories in a smaller space. A less expensive solution would be to reduce the memory demand through application tuning and optimization [32]. Furthermore, the reduction in memory requirements for a single application can be exploited in a multiprogrammed environment to support more concurrent applications. Finally, the larger the memory the more the energy demand (in both active and idle states) [29]. Due to these underlying reasons, many low-end mobile devices such as cell phones (e.g., Casio CdmaOne C452CA [2]) typically support less than 1MB of memory, of which the Java Virtual Machine (JVM) may have access to a even smaller portion.

The goal of this work is to enable the execution of Java applications using a smaller heap than that possible using current embedded JVMs. The memory requirement of a Java application is mainly shaped by the heap space required for executing the application. Existing techniques to reduce the heap space requirement include reducing the number of classes to be loaded (when classes are managed in the heap), using memory efficient data structures [40], early retirement of objects through last use analysis [35], and tuning the garbage collector (e.g., using Mark-Compact collector instead of Mark-Sweep collector) [41]. In this work, we explore the use of compression and lazy allocation combined with object partitioning as means to reduce the heap space required for executing a Java application. Reducing heap footprint can minimize the amount of active memory maintained throughout the execution and can enable the reuse of this space by another concurrent application or enable energy savings by powering down the unused memory portions. Furthermore, heap footprint reduction can also result in a smaller maximum heap size required to execute the application successfully. As a consequence, it may be possible to increase the number of applications that execute without out-of-memory exception for a given heap size.

Compression has been a potent technique for reducing memory requirements in different contexts [21, 17, 22, 26, 43]. In this paper, we present a technique that compresses objects in the heap when the current execution cannot complete normally within the given heap size. Specifically, we tune the garbage collector (GC) already present in the embedded JVM to support compression. Normally, the GC is invoked to reclaim the space occupied by garbage, i.e., the objects that are no longer needed by the application. Mark-Sweep (MS) and Mark-Compact (MC) are two garbage collection algorithms incorporated within Sun's embedded JVM (called KVM [4]), which we use in this work as a reference point. The MS collector has two phases [24, 42]: mark phase and sweep phase. During the mark phase, the collector traverses the reference tree and marks all the objects that are reachable from the roots. In the following sweep phase, the collector scans the whole heap and puts all unmarked objects into a free-table. Since this collector does not move objects after the collection, live objects and free blocks are interleaved with each other. After several allocations and collections, the heap may become so fragmented that each free block is too small to serve any object allocation request. The total size of the free blocks, however, can still be larger than the requested size. This is called the "fragmentation problem" [24, 42], due to which an application running with the MS collector usually needs larger heap space than its footprint.[1]

The MC collector addresses the fragmentation problem by supporting compaction. It also has two phases [24, 42]: mark phase and compact phase. The mark phase is the same as that of the MS collector. During the compact phase, the MC collector slides all the marked objects to one end of the heap (this operation is called the "compaction"). The free blocks, on the other hand, slide to the other end of the heap and are combined into one large free area. Since the MC collector moves objects, it needs to update each reference to an object that has been moved. The MC collector allows an application to run properly with the heap space no smaller than its footprint, which is determined by the behavior of the application.

Our Solution. In this paper, we propose a set of memory management strategies to reduce heap footprint of embedded Java applications that execute under severe memory constraints. Our first contribution is a new garbage collector, referred to as the Mark-Compact-Compress (MCC) collector, that allows an application to run with a heap smaller than its footprint. The proposed collector works in two phases: mark phase and compact-compress phase. In the mark phase, the collector not only marks the live objects, but also counts their sizes. Based on the total size of the live objects, it calculates the total size of the free space. If this size is larger than the object to be allocated, the MCC collector compacts the heap like a normal MC collector. On the other hand, if the size of the free space is smaller than that of the object to be allocated, the collector compresses all the live objects to increase available heap space. This introduces the overhead of compression and subsequent decompression when accessing a compressed object. However, since the peak heap demand occurs only during a very short period of execution, the compression overhead is not incurred frequently. Further, due to the locality of object accesses, the cost of decompression is amortized over multiple object accesses. Finally, many objects (or parts of them) are not even accessed after they are compressed since they have reached their last use (note that such objects may still be live from the collector's perspective and cannot be marked as garbage), and consequently, they are collected before any decompression.

In addition to employing compression, we also consider a heap management strategy and associated garbage collector, called MCL (Mark-Compact-Lazy Allocate), based on "lazy allocation" of "object portions". This new collector operates like the MC collector, but takes advantage of the observation that many Java applications create large objects, of which only a small portion is actually used. For example, a program may allocate a character array as the buffer when reading the user's input. Since the length of the input data is unknown at the programming time, programmers typically allocate a buffer large enough to hold the longest possible input data that can be expected from the user. However, in most cases, the actual input data may be short and the space in the buffer may be wasted. To

[1]In this paper, we define the "footprint" of an application as the maximum total size of the live objects that are in the heap simultaneously.

reduce the heap memory requirements for such programs, we break down each large array object into a set of smaller subobjects. Each subobject is "lazily allocated" upon its first write access. Therefore, the subobjects that do not contain any element actually used by the program are not heap-allocated at all, thereby saving heap space. It should be noted that, in our implementation, both breaking down large objects into subobjects and lazy allocation of subobjects are internal to JVM and transparent to the Java application being executed. In addition, we also combine MCC and MCL, and present MCCL (Mark-Compact-Compress-Lazy Allocate), which outperforms both MCC and MCL.

We implemented the proposed heap management strategies employing compression and lazy allocation using KVM [4], and compared them to two garbage collectors (MS and MC) currently used in KVM. Our experimental evaluation using a set of ten Java applications suitable for handheld devices and a zero-removal compression technique [34] shows that one of the proposed collectors reduces the peak heap demand by 35% on the average (ranging from 16% to 54%) over the MC collector. The consequent performance degradation due to compression and decompression was observed to be less than 2% on the average, over the MC collector using the same heap space. In addition, we show how our results change when we eliminate handle-based object accesses and present a garbage collector (called MCCL+) based on this. The heap management techniques proposed in this paper are very general and can be applied to other Java virtual machines and garbage collectors and can make use of different compression algorithms where available.

Roadmap. The rest of this paper is organized as follows. Section 2 gives details of our implementation. Section 3 introduces our Java benchmarks and experimental platform. Section 4 gives detailed experimental results. Section 5 investigates the impact of aggressive object compression (in comparison to the need based compression). Section 6 studies the benefits of eliminating object handles. Section 7 discusses potential research directions on this topic. Section 8 discusses related work. Finally, Section 9 concludes the paper by summarizing our major findings.

2. IMPLEMENTATION DETAILS

In this section, we present the details of our base implementation, which includes support for indirect object references, object compression and decompression, breaking down large objects into subobjects, and lazy allocation. Later in the paper we also present our enhanced implementation which eliminates object handles.

2.1 Indirect Object References

To facilitate compression/decompression of objects, in our base implementation, references to Java objects are implemented using handles (see Figure 1). Specifically, each object has a handle in the handle pool. Each handle has two components: a pointer to instance data and a pointer to class data in the class area. An object reference is actually a native pointer to a handle in the handle pool. An advantage of this scheme is that, when an object is moved, it eliminates the necessity of updating every reference to this object, which may be scattered in different locations in the runtime data area. The main drawback is that each access to an object's instance data requires dereferencing two pointers.

In our implementation, allocation of a Java object involves two steps: (1) allocate a handle in the handle pool; (2) allocate the heap space for the data of the object. When the handle pool is used up, the GC is invoked. The handles of the objects that have been collected are returned to the handle pool. If the free space in the handle pool is smaller than a given threshold (T bytes) after the collection,

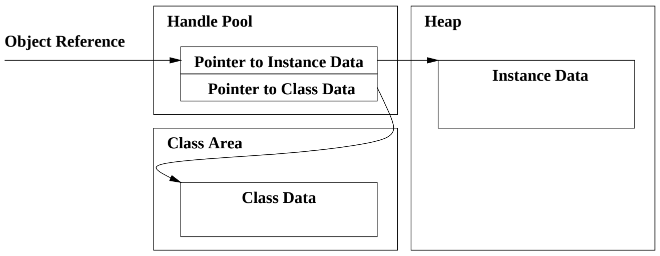


Figure 1: Referencing an object through a handle. In this implementation, the allocation of an object involves two steps: (1) allocate a handle in the handle pool; (2) allocate the heap space for the data of the object. In the MC collector, the handle pool is a part of the permanent space (which is not garbage collected).

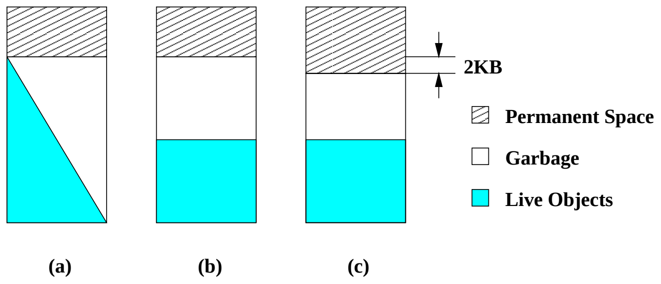


Figure 2: Expansion of the permanent space. (a) Before garbage collection. (b) After garbage collection — the live objects are compacted into one end of the heap, and the free blocks are slided to the other end and are combined into one large free area. (c) Expanding the permanent heap space.

the handle pool is expanded by C bytes. The expansion of the handle pool is to avoid frequent garbage collections due to small handle pool size. The threshold, on the other hand, is set to prevent the handle pool from growing too fast. Based on our experience, C and T are set to 1/64 and 1/32 of the heap size, respectively. Since each handle has 8 bytes, the total size of the handle pool is bounded by $8M + T + C$, where M is the maximum number of live objects in the heap.

Since the handles cannot be moved, the handle pool is considered as part of the permanent space. The permanent space is the memory space that contains the data whose lifetimes last until the application terminates. The permanent space is never garbage collected in this study. This space expands in 2KB chunks (the default value in KVM) when it is used up. Expansion of the permanent space involves garbage collection and, if necessary, compacting (or compressing) live objects to one end of the heap, as shown in Figure 2. It should be noted that, in the implementation of the MS collector, there exists no separate permanent space since this collector does not compact the heap. All the permanent data are allocated in the heap and scanned by the collector, although they are never collected.

2.2 Compression

MCC collector compresses objects when compaction cannot provide enough space for the new object. In principle, our approach can work with any compression/decompression algorithm. However, for the best results, a compression/decompression algorithm should satisfy three requirements: (1) the compressor should have a

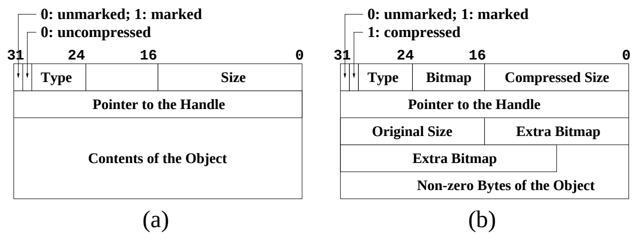


Figure 3: (a) Format of an uncompressed object. (b) Format of a compressed object. In both formats, the first 8 bytes (headers) are used by garbage collector for management purpose. Each bit in the bitmap in (b) corresponds to a byte of the object's data in the uncompressed format.

good compression ratio;[2] (2) both compression and decompression should be fast; and (3) neither the compressor nor the decompressor should use a large working area. In this paper, we used a "zero removal" compression scheme, which is based on the observation that a large portion of memory locations manipulated by an application contains only zeroes [34]. The uncompressed and compressed object formats are shown in Figures 3(a) and 3(b), respectively. In Figure 3(b), the first eight bytes of each object (i.e., the object header) are not compressed and the compressed object contains a bitmap and an array of non-zero bytes. Each bit in the bitmap corresponds to a byte of the object's data in the uncompressed format. A 0-bit indicates that the corresponding byte is all zero and this byte is not stored in the compressed format. A 1-bit, on the other hand, indicates a non-zero byte and that this byte is kept in the array of non-zero bytes in the compressed format. Bits 16 through 23 of the first word in the header are not used in the uncompressed format. In the compressed format, however, they contain the first eight bits of the bitmap. For an object whose size is larger than eight bytes, the extra bits of the bitmap are stored right after the object's "original size" field. Following the bitmap is the array of non-zero bytes of the object.

When our collector decides to compress objects, it scans the whole heap and compresses each object that has not been compressed so far. In our current implementation, only the object instances and arrays that are created by the Java application are compressed. The data structures internal to the implementation of JVM remain uncompressed. In other words, for them, the compression works in the same way as compaction. Figure 4 illustrates the heap compression process. We maintain two pointers: *source* and *target*. *Source* points to the next object to be compressed (or to be compacted, for the internal data structures), and *target* points to the first free location in the heap.

Figure 5 shows the algorithm for compressing an object. An example application of this algorithm is illustrated in Figure 6. Step 0 shows the state right before an object is to be compressed. Pointer *target* points to the first free location; pointer *source* points to the first byte of the object to be compressed. At step 1, pointer t is initialized to the location where the compressed object will be stored; pointer s is initialized to the first byte to be compressed; and pointer p is initialized to the first byte where the non-zero byte array will be stored. The bitmap size is calculated using the size of the object. At step 2, the first bit of the bitmap is set to 1 as the first byte of the object (which is pointed to by s at step 1) is non-zero. Since $s < p$, the byte pointed to by s is temporarily stored in the

[2]"Compression ratio" is the ratio between the size of the original data and the size of the compressed data.

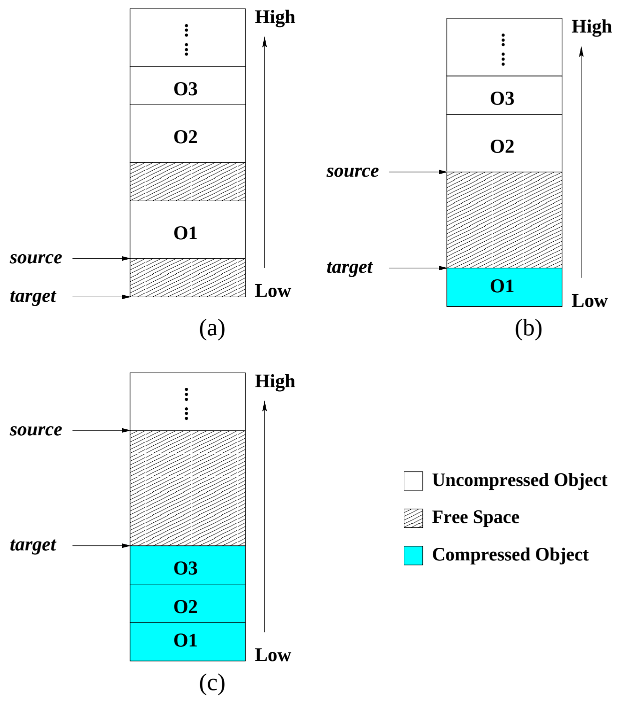


Figure 4: Compressing the heap. *source*: the next object to be compressed (or to be compacted, for the internal data structures). *target*: the first free location in the heap. (a) Initial state. (b) After compressing O1. (c) After compressing O1, O2, and O3. Note that, after compressing each object, both *source* and *target* are updated.

buffer. Since this byte is non-zero, p is increased by 1. At step 3, s is now pointing to a zero byte. Therefore, at the next step, the third bit of the bitmap is set to 0 and p is not increased. At step 6, s catches p, and from this point on, no more bytes are placed into the buffer. Instead, all the non-zero bytes are copied to the location pointed to by p. At the last step, all the bytes in the buffer are copied to their destination locations. In our implementation, we used unmarked objects (whose addresses should be above that of the object being compressed) to hold the buffer. Since the buffer is accessed sequentially, the space allocated for the buffer does not need to be contiguous. Several unmarked objects can be chained up using links in case that the buffer becomes too large to be accommodated in a single unmarked object. It should be noted that, for most of the objects, the distance between the pointers *source* and *target* (Figure 4) is so large that even the initial value of p is smaller than s. Therefore, the buffer is rarely used during the compression process. A rare case occurs when the compressed size of the object is larger than its original uncompressed size. This does not happen frequently (on average, only 0.3% of the objects in our applications caused such an expansion) and does not impact the overall memory consumption significantly. However, it still needs to be addressed since there may be no space for the object to expand. In our implementation, the compressor checks if the pointer p exceeds the starting address of the next object whenever p is increased. If this happens, the compressor stops compression and uses the data in the buffer and the bitmap to recover the object.

```
byte heap[];
void **compress**(int source, int target)
{
    s = source; t = target;
    p = t + $\lceil length\_of\_the\_object/8 \rceil$;
    while(s - source $< length\_of\_the\_object$) {
        if(h[s]==0) {
            set the current bit in heap[t] to 0;
        } else {
            set the current bit in heap[t] to 1;
            if(p$\leq$s)
                heap[p]=heap[s];
            else
                append heap[s] to the buffer;
            p++;
        }
        s++;
        increase t if necessary
    }
    copy all the bytes from the buffer
    to the locations beginning from t;
}
```

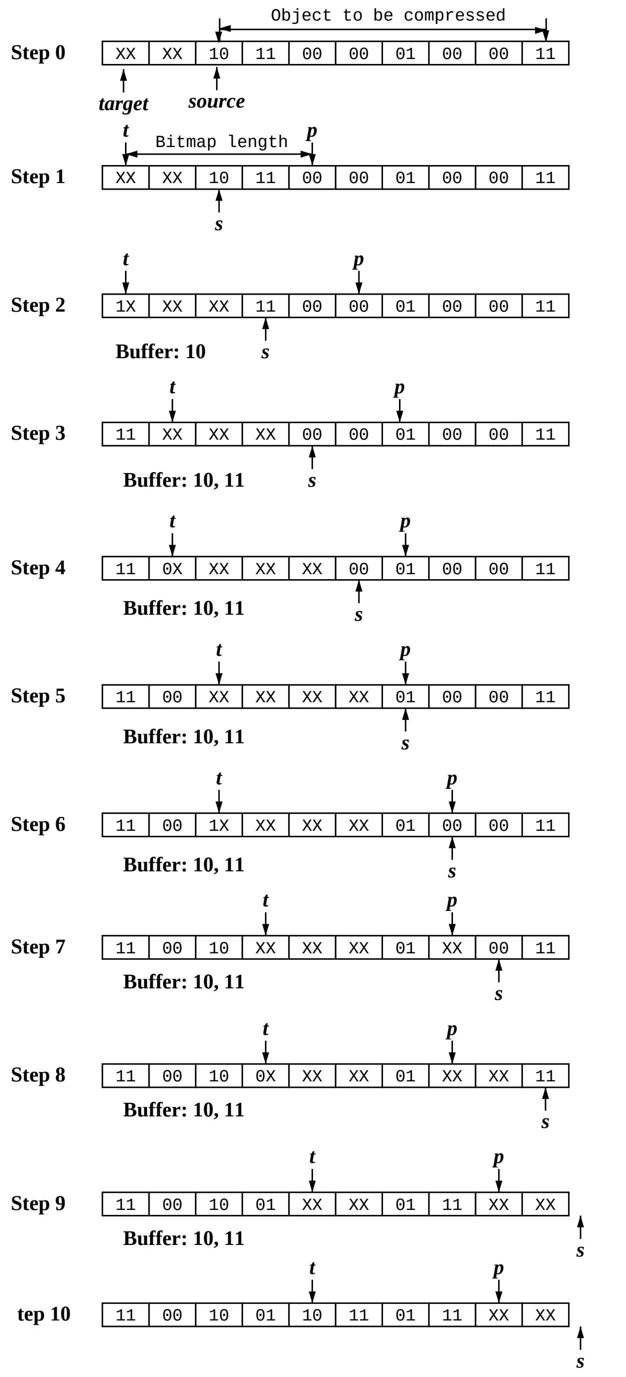
Figure 5: Zero-removal compression algorithm. This algorithm compresses a single object. To show the core logic of our algorithm, implementation details are omitted.

2.3 Decompression

If a compressed object needs to be accessed, the decompressor is invoked. Whenever an object is accessed, the virtual machine checks if the object has been compressed. For this purpose, we use the highest order bit of the pointer to the instance data of each handle as a flag. For an uncompressed object, this bit is zero since current JVMs for memory constrained devices do not use an address space larger than 2GB. When an object is compressed, we set this bit to 1. To access an object, the virtual machine first loads the pointer to the instance data from the handle into a register (the cost for this instruction is captured in our experimental results). After that, the virtual machine checks if the first bit of the register is zero. If the bit is zero, the object does not need to be decompressed. Otherwise, the object needs to be decompressed first. This checking requires two instructions: one comparison and one branch. Note that the comparison instruction uses the contents of the register and does not involve any extra main memory access, and that the branch instruction is highly predictable (since most objects are not compressed). Therefore, one can expect the overhead associated with checking the compression status of objects to be small.

The decompression process is illustrated in Figure 7 and our decompression algorithm is given in Figure 8. The decompressor first allocates a free block that is large enough to hold the object to be accessed in uncompressed format. If we fail to allocate a free block successfully, the GC is invoked. The decompressor, if successful in allocating a free block, decompresses the object into the block. It also updates the pointer in the handle. The object in the compressed format, however, is discarded and is collected in the next invocation of the GC.

Decompression also happens during the mark phase of garbage collection. In the mark phase, the collector traverses the reference tree. When the collector visits a compressed object, it first checks the object's class data to see if this object contains any reference fields. If this is the case, the collector decompresses the object to retrieve the contents of the reference fields. Note that the decompression in the mark phase is different from the decompression that happens during the application execution in that the former does not keep the data of the decompressed object. Specifically, when



Object to be compressed
Step 0
XX XX 10 11 00 00 01 00 00 11
target
source
Step 1
t
Bitmap length
p
XX XX 10 11 00 00 01 00 00 11
s
Step 2
t
p
1X XX XX 11 00 00 01 00 00 11
Buffer: 10
s
Step 3
t
p
11 XX XX XX 00 00 01 00 00 11
Buffer: 10, 11
s
Step 4
t
p
11 0X XX XX XX 00 01 00 00 11
Buffer: 10, 11
s
Step 5
t
p
11 00 XX XX XX XX 01 00 00 11
Buffer: 10, 11
s
Step 6
t
p
11 00 1X XX XX XX 01 00 00 11
Buffer: 10, 11
s
Step 7
t
p
11 00 10 XX XX XX 01 XX 00 11
Buffer: 10, 11
s
Step 8
t
p
11 00 10 0X XX XX 01 XX XX 11
Buffer: 10, 11
s
Step 9
t
p
11 00 10 01 XX XX 01 11 XX XX
Buffer: 10, 11
s
tep 10
t
p
11 00 10 01 10 11 01 11 XX XX
s

Figure 6: Example compression. X: "don't care" bit. For illustrative purposes, each byte is assumed to have only 2 bits. Pointer *target* points to the first free location; pointer *source* points to the first byte of the object to be compressed. At step 1, pointer *t* is initialized to the location where the compressed object will be stored; pointer *s* is initialized to the first byte to be compressed; and pointer *p* is initialized to the first byte where the non-zero byte array will be stored.

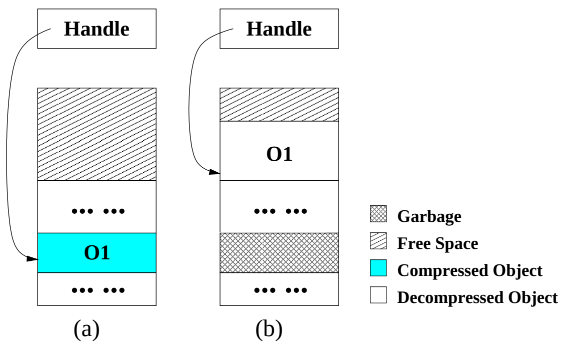


Figure 7: Decompressing object O1. (a) Before decompression. (b) After decompression. The decompressor first allocates a free block and then decompresses the object into the block. It also updates the pointer in the handle.

```
byte heap[];
void decompress(int source, int target)
{
    b = source; // set b to the first byte of the bitmap
    s = source + $\lceil length\_of\_the\_object/8 \rceil$;
        // set s to the first non-zero byte
    t = target; // set t to the first byte of the target location
    while(t - target $< length\_of\_the\_object$) {
        if(current bit is 0)
            heap[t++] = 0;
        else
            heap[t++] = heap[s++];
        increase b if necessary
    }
}
```

Figure 8: Decompression algorithm. This algorithm is invoked at object access time and decompresses a single object. To show the core logic of our algorithm, implementation details are omitted.

the collector encounters a compressed object that has at least one reference field, the collector scans the object and decompresses it field by field. When each field is decompressed, the collector first checks to see if this field contains a reference and, if so, marks the referenced object. The decompressed field is discarded immediately. Therefore, the decompression in the mark phase does not involve any allocation.

2.4 Breaking Down Large Objects

It is not efficient to decompress a whole object when only a few fields of the object are accessed. This is particularly true when the object in question is very large. Decompressing the whole object not only increases memory requirements, but also slows down the application due to longer decompression time. To address this problem, we propose to break large objects into smaller "subobjects", as shown in Figure 9. Specifically, an object whose size is larger than a given threshold (1.5KB in this work) is broken down into a set of smaller subobjects (each with a maximum size of 1KB). Each subobject is compressed and decompressed independently. It should be noted that, since Java object instances are not likely to be larger than 1KB, in this work, only Java arrays are considered to be broken down into smaller portions.

In our implementation, upon accessing an element of a Java array, the virtual machine first checks if this array has been broken down. If this is the case, the virtual machine uses the index of the element and the size of the corresponding subobjects to calculate the index of the subobject that contains the element and the intra-subobject offset for the element. If the subobject is in compressed format, JVM also invokes the decompressor to decompress the subobject.

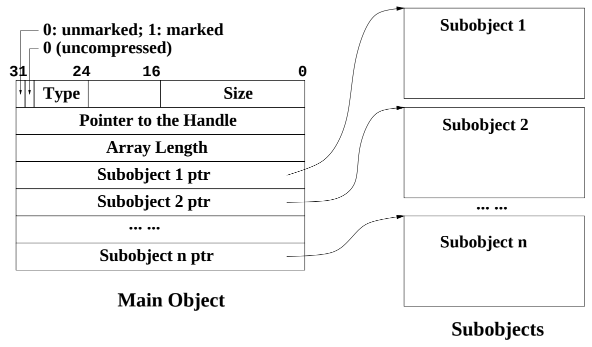


Figure 9: Breaking down a large array into subarrays. In this way, each subarray (subobject) can be allocated independently.

2.5 Lazy Allocation

Our observation is that many Java applications do not access different portions of objects uniformly. That is, some fields are accessed much more frequently than the others. As a result, heap memory can be saved if different portions of a given object are allocated in an on-demand basis. In other words, it may be beneficial if we do not allocate a portion of the object unless that portion is actually accessed. Shaham et al. [35] studied a similar strategy at the whole object level; that is, no heap space is allocated unless the object is used. Our lazy allocation strategy differs from [35] in two important aspects. First, instead of whole objects, we consider different portions of objects; that is, our approach is finer granular. Second, unlike the approach in [35], our approach is entirely transparent to the application execution. In order not to introduce too much runtime overhead, we applied lazy allocation only to the large arrays. In our implementation, when the bytecode "NEWARRAY" is encountered, the allocator checks if the size of this array is larger than a pre-set threshold. If this is not the case, the array creation proceeds as usual. Otherwise (i.e., if this array needs to be broken-down into subarrays), the allocator allocates a main object for this array (see Figure 9). The subobjects (subarrays), however, are not allocated at this time. Instead, the pointers to the subobjects are set to null. When an element of such an array is later accessed, the virtual machine checks to see if the pointer to the subobject that contains that array element is null. If the pointer is null and the current access is write, JVM allocates the heap space for the subobject, sets each element in this subobject to the uninitialized value (as defined in JVM specification [30]), and then updates the value of the element that is being accessed. In other words, each subobject is lazily created upon its first write access. A null pointer indicates that the elements in the subobject have not been initialized by the application since the array has been created. Therefore, for each read access to a null subobject, JVM returns a pre-defined uninitialized value according to the type of the element.

3. BENCHMARKS AND EXPERIMENTAL SETUP

In this paper, we experimented with six different garbage collection strategies listed in Table 1. The first two of these strategies (MS and MC) are the default Mark-Sweep and Mark-Compact

Scheme	Compaction?	Compression?	Reference	Breaking Down Large Objects?	Lazy Allocation?
MS Mark-Sweep	No	No	Direct	No	No
MC Mark-Compact	Yes	No	Direct	No	No
MCL Mark-Compact-Lazy Allocate	Yes	No	Direct	Yes	Yes
MCC Mark-Compact-Compress	Yes	Yes	Handle	No	No
MCCL Mark-Compact-Compress-Lazy Allocate	Yes	Yes	Handle	Yes	Yes
MCCL+ Mark-Compact-Compress-Lazy Allocate	Yes	Yes	Direct	Yes	Yes

Table 1: The garbage collection strategies evaluated in this paper. The first two of these strategies (MS and MC) are the default Mark-Sweep and Mark-Compact collectors, whereas the remaining ones are our strategies.

collectors, which are currently employed in Sun's KVM [4, 6], whereas the remaining ones are our strategies. These strategies differ from each other in how they reference an object (direct or handle based), whether they break down large objects into smaller subobjects, or whether they employ lazy allocation. In the next section (Section 4), we conduct an experimental evaluation of MCC, MCL, and MCCL. The detailed discussion of MCCL+ will be presented later in Section 6. *The important point to emphasize here is that all our compression based collectors use compression only in object allocation and only when it is not possible to continue execution without performing compression (as explained in Section 1). In other words, we evaluate a need based object compression strategy. Later in Section 5 we evaluate a more aggressive object compression strategy as well.*

To evaluate different garbage collection strategies listed in Table 1, we used a set of ten Java applications as benchmarks. These benchmarks represent typical applications running on handheld devices where memory budgets are very limited. Brief descriptions of our benchmarks are given in Table 2. As can be seen, our benchmark suite includes utility applications as well as game programs.

Table 3 presents memory allocation data for these Java benchmarks. For each benchmark, the second column of this table gives the total numbers of Java objects allocated by the benchmark throughout its execution (including both Java objects and arrays), and the third column gives the average and maximum object sizes. The fourth column shows the maximum size of the objects that are live in the heap simultaneously, and the next column shows average heap occupancy (i.e., the percentage of heap that is occupied by live objects at a given time), when each benchmark is executed using the minimum heap size that allows it to run without an out-of-memory exception. We can see that, even with the minimum heap size that allows the benchmark to run, on the average (across all benchmarks), only 65.57% of the heap is occupied at a given time. Finally, the sixth column gives the overall execution time and the last column shows the GC execution time (to obtain the numbers in these last two columns, we executed each benchmark with the minimum heap size that allows it to run without an out-of-memory exception).

4. EXPERIMENTAL RESULTS

4.1 Reduction in Heap Space Demand

We expect our new garbage collection strategies to be useful in two aspects. First, we expect our strategies to reduce the minimum heap size necessary to execute an application without out-of-memory exception. Second, our strategies reduce the heap occupancy. That is, at a given time, our approach reduces the heap memory requirements of the application being executed. In this subsection, we provide experimental data to quantify these two benefits.

Table 4 gives the minimum heap sizes for each benchmark to run without out-of-memory exception using different garbage collectors. In this section, we mainly focus on MS, MC, MCL, MCC, and MCCL, and postpone the discussion of MCCL+ to a later section. The first part of this table (that is, the columns two through seven) gives the absolute heap sizes in KBs, whereas the second part gives the values (heap sizes) "normalized" with respect to that of the MC collector. Our first observation from the results in Table 4 is that, compared to the MC collector, the MS collector requires 47.9% more heap space on the average (i.e., across all our benchmarks). This is a direct result of the fragmentation problem. We also observe that both lazy allocation (in conjunction with breaking large objects into smaller subobjects) and object compression help reduce the applications' heap memory requirements. More specifically, on the average, MCL and MCC brought down the heap memory requirements of our benchmarks by 9.5% and 10.8%, respectively, with respect to MC (the average reduction with respect to MS is around 40%). Combining them in MCCL results in even more heap memory space savings (21% on the average). An exception is Sfmap, where MCC requires more heap memory than MC. This is because of two main reasons. The first is that the handle pool requires extra space. The second reason is that Sfmap allocates many more large objects as compared to other benchmarks in our experimental suite. As discussed earlier, when a Java object is being decompressed, JVM has to maintain the object in both compressed and uncompressed formats until the decompression is completed. Holding a large object in both formats simultaneously increases the pressure on the heap memory. Actually, this is one of the motivations to break down large Java objects into smaller ones. In fact, breaking down large objects enhances the effectiveness of object compression for this benchmark. Specifically, one can observe that although compression alone (MCC) does not seem to be very useful for Sfmap, combining it with object breakdown (MCCL) brings an extra 27KB heap space saving over the MCL collector. To summarize, our compression and lazy allocation based strategies reduce the minimum heap size demand of Java applications; that is, they allow applications to execute with smaller heap sizes.

We now analyze the second benefit of our strategies, namely, reducing the heap occupancy during the course of execution. Figure 10 shows the total size of the live objects in the heap over time. Only four benchmarks are shown here since the trends for other benchmarks are similar to those presented here. We can observe

Benchmark	Description	URL
Auction	Client for online auction	An example comes with MIDP 2.0 reference implementation [6]
Calculator	Numeric calculator	www.spruce.jp/freemidlets/
JBrowser	WAP browser	www.jataayusoft.com/
JpegView	JPEG image renderer	www.jshape.com/midp/index.html
ManyBalls	Multithreaded game	An example comes with MIDP 2.0 reference implementation [6]
MDoom	3D shooting game	www.jshape.com/midp/index.html
PhotoAlbum	Digital photo album	An example comes with MIDP 2.0 reference implementation [6]
Scheduler	Personal monthly scheduler	holycow.tripod.co.jp/cooldownboy/
Sfmap	Interactive digital map	www.jshape.com/midp/index.html
Snake	Game	An example comes with MIDP 2.0 reference implementation [6]

Table 2: Java benchmarks used in our experiments. The second column gives a brief description, and the third column shows where the benchmark can be found. Note that our benchmark suite includes utility applications as well as game programs.

Benchmark	Total Number of Allocated Objects	Object Size Average (Maximum)	Total Size of Live Objects in the Heap		Execution Time (Seconds)	GC Time (Seconds)
			Maximum	$\frac{\text{Average}}{\text{Maximum}}$		
Auction	5123	39 (8492)	84596B	68.94%	96.08	4.85
Calculator	11250	26 (1036)	39824B	67.80%	65.94	2.19
JBrowser	16160	56 (12012)	229432B	72.37%	338.37	21.85
JpegView	10199	44 (8972)	86524B	88.88%	417.35	50.03
ManyBalls	2090	35 (1036)	35088B	77.19%	461.91	1.28
MDoom	1319	61 (16396)	126408B	40.98%	500.56	1.43
PhotoAlbum	3864	83 (4260)	54388B	55.26%	66.71	4.30
Scheduler	10042	66 (1036)	35464B	76.52%	253.09	5.77
Sfmap	6599	27 (2460)	166224B	57.90%	81.37	3.95
Snake	1776	29 (1036)	40072B	70.59%	68.96	1.53
Average:	6842	47 (5674)	89802B	65.57%	216.03	9.72

Table 3: Heap-related behavior of our benchmarks. The numbers in the second column include both Java object instances and arrays. The third column gives the average and maximum object sizes. The fourth column shows the maximum size of the objects that are live in the heap simultaneously, and the next column shows average heap occupancy when each benchmark is run using the minimum heap size that allows it to run without out-of-memory exception (increasing the heap size reduces the percentage heap occupancy). Execution times and garbage collection times are measured by running each benchmark with minimum heap size that allows the benchmark to run without out-of-memory exception.

from this figure that peak of the heap memory demand does not occur frequently and that each peak lasts for only a short period of time. One can also see that, at a given time, our collectors reduce the pressure on the heap space as compared to the MC collector. Note that this reduction in heap occupancy might be important in different contexts. For example, in a multiprogrammed environment, a reduction in heap space can allow other applications to utilize the unused memory. Alternately, the unused memory parts can be placed into a low power operating mode [14] to reduce memory energy consumption. Quantifying the impact of heap space reduction from these two angles is in our future agenda.

4.2 Compression/Decompression Behavior

Analyzing the number of compressions and decompressions is very important as it has a direct impact on performance (execution cycles). Table 5 gives the number of compressions and decompressions for each benchmark running with MCCL when the entire execution is considered. Note that if the same object is compressed (decompressed) N times, we counted it as N compressions (decompressions). In these experiments, each application was run with the minimum heap size that allowed it to execute without out-of-memory exception (as shown in the sixth column of Table 4). Comparing the number of compressions (the second column in Table 5) with the total number of created objects (the second column in Table 3), one can observe that only a small percentage of our objects were compressed (9.1% on the average). This is a direct result of our compression policy: we compress objects only when we really need to compress them; that is, at the peaks of the heap space demand. And, since such peaks do not occur very frequently and each peak lasts for only a short period of time, we do not perform frequent object compressions. We also observe from Table 5 that only a small percentage (22.36% on the average) of all compressed objects are decompressed. This is an interesting result and indicates that most compressed objects are not used subsequently by the application, although they are still live at the moment they are compressed (from the GC's perspective). The last three columns of this table give the number of objects that have been decompressed only N ($N > 0$) times. We find that no object has been decompressed more than three times. We can also see from these results that most of our objects have been decompressed only once or twice. In fact, among the objects that have ever been decompressed, an overwhelming majority (82.5%) have been decompressed only once. That is, after they have been compressed, they have been decompressed only once, and subsequently (after, possibly, several accesses), they have become garbage. Thus, the numbers in these last three columns explain the low percentage values shown in the column four of Table 5.

It is also important to compare MCC and MCCL from the perspective of the number of object compressions and decompressions. Such a comparison is given in Table 6, which shows the same information as in Table 5, except that the applications are run here using the minimum heap size that allows the MCC collector to execute without an out-of-memory exception (in contrast, the results in Table 5 were obtained by running applications using the minimum heap size that allows the MCCL collector to execute without an out-of-memory exception). The results given in Table 6 clearly indicate the importance of lazy allocation and breaking down large objects into subobjects. Specifically, using MCCL instead of MCC

	Minimum Heap Size (KB)						Normalized against MC (%)				
Benchmark	MS	MC	MCL	MCC	MCCL	MCCL+	MS	MCL	MCC	MCCL	MCCL+
Auction	128	83	76	72	62	58	154.2	91.6	86.8	74.7	69.9
Calculator	55	40	40	34	34	32	138.5	100.0	85.0	85.0	80.0
JBrowser	260	226	196	195	164	157	115.0	86.7	86.2	72.6	69.5
JpegView	127	85	85	79	77	64	149.4	100.0	92.9	90.6	75.3
ManyBalls	57	35	35	31	31	29	162.9	100.0	88.6	88.6	82.9
MDoom	178	124	71	114	76	57	143.5	57.3	91.9	61.3	46.0
PhotoAlbum	96	55	55	50	50	46	174.5	100.0	90.9	90.9	83.6
Scheduler	56	37	36	32	32	31	151.4	97.3	86.5	86.5	83.8
Sfmap	292	162	118	175	91	78	118.5	72.8	108.0	56.2	48.1
Snake	72	42	42	35	35	33	171.4	100.0	83.3	83.3	78.6
Average:	132	90	75	81	65	59	147.9	90.5	89.2	79.0	65.6

Table 4: Minimum heap sizes to execute benchmarks without an out-of-memory exception. The first part of this table (that is, the columns two through seven) gives the absolute heap sizes in KBs, whereas the second part gives the values (heap sizes) "normalized" with respect to that of the MC collector.

Benchmark	Total Number of Comp.	Decompressions		Number of Objects Decompressed N times		
		Total Number	% of Total Comp.	$N = 3$	$N = 2$	$N = 1$
Auction	583	183	31.39%	4	22	127
Calculator	226	46	20.35%	0	5	36
JBrowser	1393	431	30.94%	0	135	161
JpegView	1725	433	25.10%	0	23	387
ManyBalls	264	83	31.44%	0	18	47
MDoom	0	0	N/A	0	0	0
PhotoAlbum	235	54	22.98%	0	0	54
Scheduler	172	36	20.93%	0	0	36
Sfmap	1431	92	6.43%	0	0	92
Snake	234	38	16.24%	0	0	38
Average:	626	140	22.36%	0.4	20.3	97.8

Table 5: The number of compressions and decompressions using MCCL. Each benchmark was run with the minimum heap size that allowed it to complete without an out-of-memory exception (see the sixth column of Table 4). The fourth column gives the number of decompressions as a percentage of the number of compressions. The last three columns indicate that most of the Java objects are decompressed only once or twice.

Benchmark	Total Number of Comp.	Decompressions		Number of Objects Decompressed N times	
		Total Number	% of Total Comp.	$N = 2$	$N = 1$
Auction	530	131	24.72%	0	131
Calculator	226	46	20.35%	5	36
JBrowser	939	261	27.80%	0	261
JpegView	1926	867	45.02%	228	411
ManyBalls	264	83	31.44%	18	47
MDoom	365	207	56.71%	14	179
PhotoAlbum	235	54	22.98%	0	54
Scheduler	172	36	20.93%	0	36
Sfmap	1429	90	6.30%	0	90
Snake	234	38	16.24%	0	38
Average:	632	181	28.69%	26	128

(a) MCC

Benchmark	Total Number of Comp.	Decompressions		Number of Objects Decompressed N times	
		Total Number	% of Total Comp.	$N = 2$	$N = 1$
Auction	530	131	24.72%	0	131
Calculator	226	46	20.35%	5	36
JBrowser	939	261	27.80%	0	261
JpegView	0	0	N/A	0	0
ManyBalls	264	83	31.44%	18	47
MDoom	0	0	N/A	0	0
PhotoAlbum	235	54	22.98%	0	54
Scheduler	172	36	20.93%	0	36
Sfmap	0	0	N/A	0	0
Snake	234	38	16.24%	0	38
Average:	260	65	24.96%	2	60

(b) MCCL

Table 6: The number of compressions and decompressions for (a) MCC and (b) MCCL. Each benchmark was run with the minimum heap size that allowed it to execute without out-of-memory exception using MCC (see the fifth column of Table 4). One can see that using lazy object allocation and breaking down large objects into smaller ones helps reduce the number of compressions/decompressions significantly over the pure compression based strategy.

results in a 59% (64%) reduction in the number of compressions (decompressions).

Since MCCL is the most effective of all the strategies evaluated so far in reducing the heap memory demand, we wanted to study its behavior more carefully. Figures 11 and 12 present the heap memory usage of our benchmarks under the MCCL collector. Each benchmark was run using three different heap sizes: (1) the minimum heap size required by MCCL (the sixth column of Table 4), (2) the minimum heap size required by MC (the third column of Table 4), and (3) 150% of the size in (2). We use heap size (2) basically to compare the behavior of MCCL with MC in the course of execution, and (3) to demonstrate the behavior of MCCL when there exist plenty heap space. Each graph shown in Figure 11 and Figure 12 has three curves, labeled as "Overall", "Live," and "Compressed." "Overall" represents the heap usage at a given point during execution, i.e., the total size of the live objects (including both compressed and uncompressed objects) and garbage in the heap. It should be noticed that each drop in the "Overall" curve indicates an invocation of the GC. The curve labeled "Live" corresponds to the total size of live objects, including both compressed and uncompressed.[3] Finally, the curve labeled "Compressed" represents

[3]One may expect that an increase in Live curve should always be accompanied with an increase in the Overall curve in Figures 11 and 12. However, this is not true due to the allocation of stack frames. In handheld devices, due to memory constraints, KVM does not use a separate stack space for each Java thread. Instead, it allocates stack chunks in the heap. Each stack chunk is 520 bytes and contains one or more stack frames. A stack chunk may become garbage when the corresponding method returns. A garbage stack chunk is detected immediately without invoking the GC. Garbage stack chunks are put in a free stack chunk table. When the application needs a new stack chunk, if the free stack chunk table is not empty, KVM allocates the chunk from this table. In this case, we observe an increase in the Live curve trend while the Overall curve remains unchanged. The free chunk table is emptied after each in-

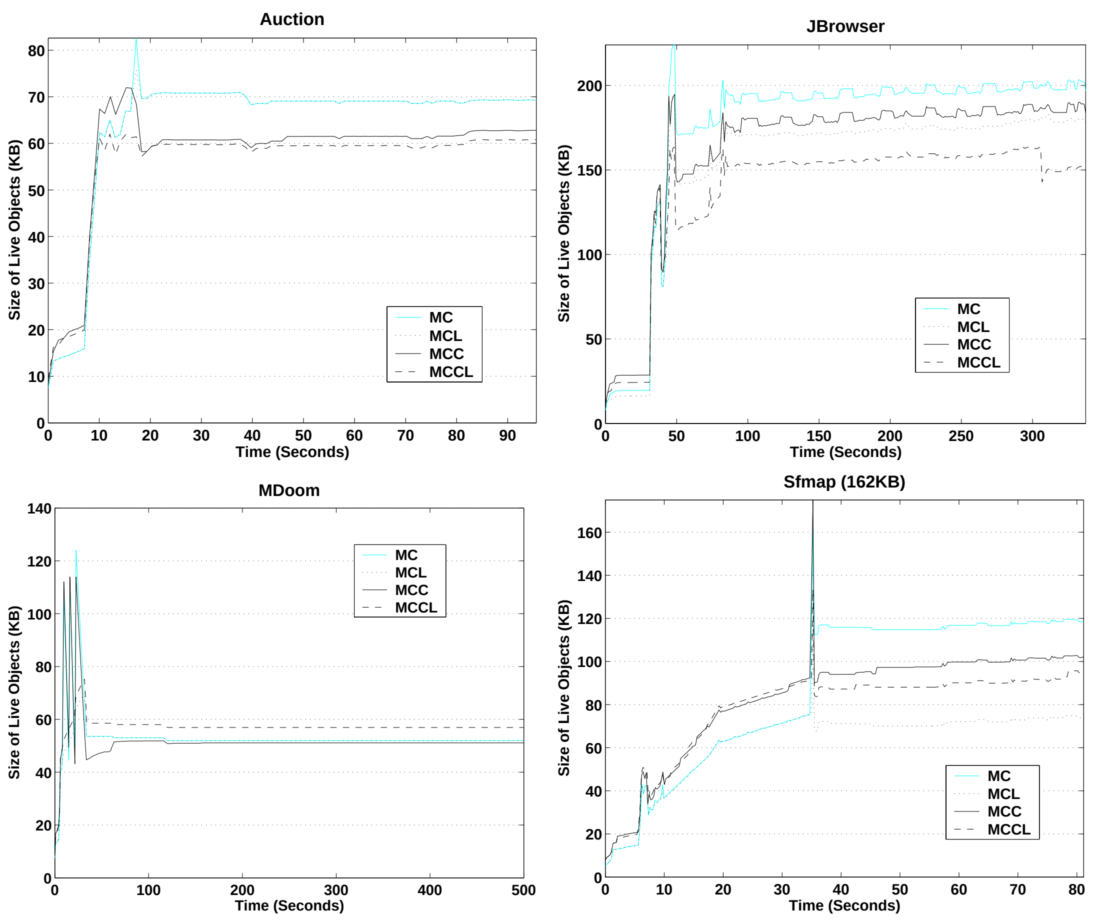


Figure 10: The sizes of the live objects for four of our benchmarks. One can observe that, at a given time, our collectors reduce the pressure on the heap space as compared to the MC collector. Similar behavior is also observed with remaining benchmarks in our experimental suite.

the total size of all compressed objects. This curve climbs up when objects are compressed and drops when a compressed object becomes garbage or is decompressed. Figures 11 and 12 clearly indicate that compressions are performed only at peaks of heap memory demands. As discussed earlier, these peaks do not occur very frequently and each peak, when occurs, does not last very long.

4.3 Performance Impact

While compressing heap objects is beneficial from the memory usage perspective, it is also important to consider its impact on performance. Figure 13 gives the runtime overheads incurred by MCCL. The overheads in this figure are given as percentages of the execution time of MC assuming an "infinite" heap space (in reality, to calculate the time with the infinite heap space, we calculated the time with the finite heap space and deducted the time taken by the GC). We normalized the performance overhead to the ideal execution time (i.e., the execution time with infinite heap space) due to two reasons. The first is that, ideally, given the same application and the same input, changing the heap size can only change the time spent within the GC; the time spent for executing the bytecodes is equal to the ideal execution time regardless of the actually size of the heap. By normalizing the overheads with respect to the ideal execution time, we are able to compare the overheads across different heap sizes; this enables us to study the impact of the heap size. Second, the MC collector cannot run with the minimum heap size that allows MCCL to run. In this case, it is difficult to normalize the overhead with respect to the execution time of MC with the same heap size.

The overhead in Figure 13 is divided into several components. "Lazy Access" represents the time overhead due to lazy allocation. "Indirect Reference" corresponds to the overhead due to using object handles. "Compression" is the time spent in compression and "Decompression" is the time spent in decompression when accessing objects. "GC Decompression" is the time spent in decompression during GC (when traversing the heap). The component denoted as "Check" represents the time spent at each object access to check whether the object is in the compressed format or not. Finally, "Other GC Time" is the time spent in collecting garbage (note that the infinite heap configuration does not use garbage collection). Our first observation from the graph in Figure 13 is that, on the average, working with MCCL brings a 9.1% performance overhead as compared to the ideal heap scenario. It should be noted, however, that only 3.5% overhead is actually due to factors other than the time spent in performing garbage collection (that is, the "Other GC Time" component); and, this last component should exist (in varying magnitudes) with any limited size heap (i.e., it is not due

vocation of the GC since all the garbage stack chunks have been collected.

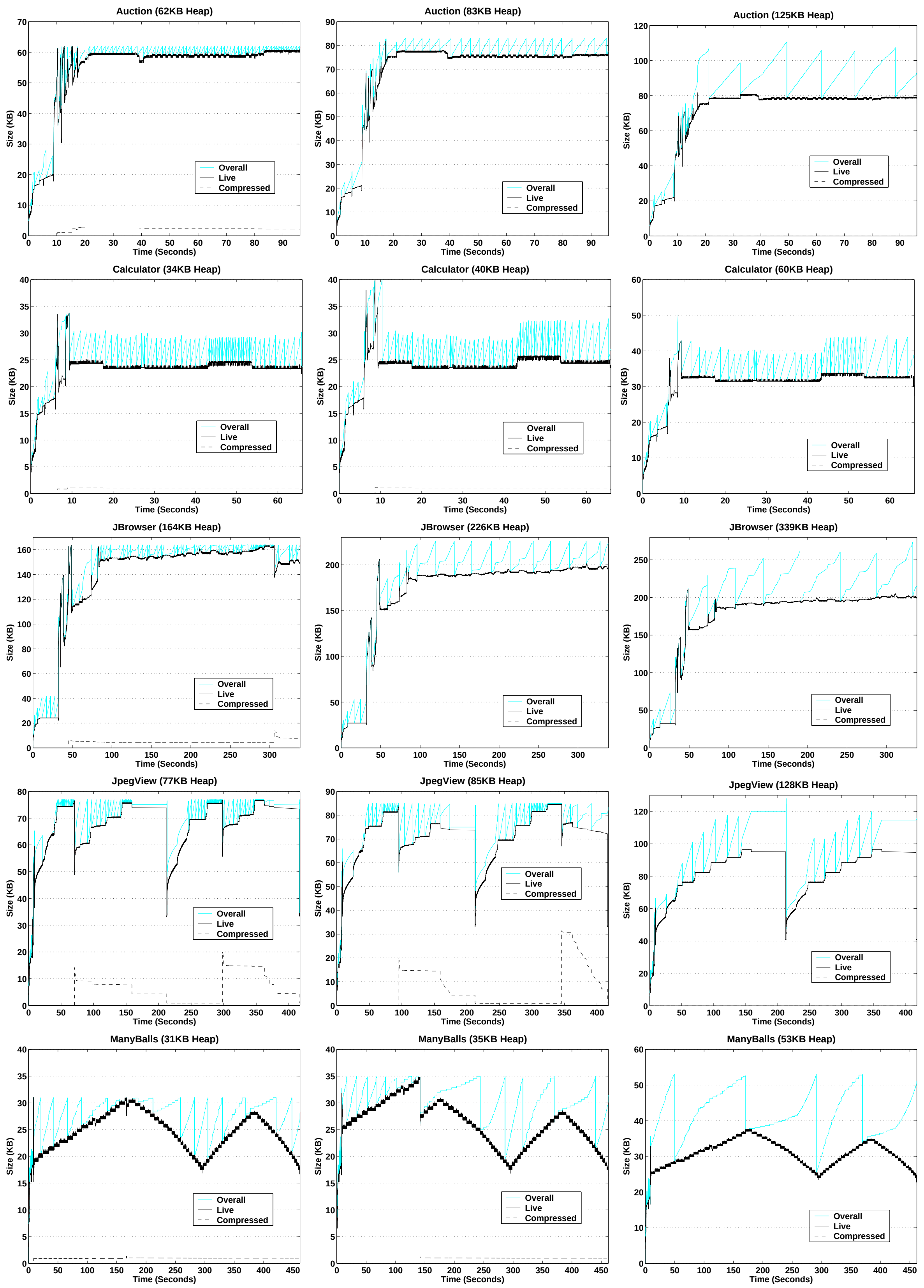


Figure 11: Heap memory usage of MCCL. "Overall" represents the heap usage at a given point during execution, i.e., the total size of live objects (including both compressed and uncompressed objects) and garbage in the heap. The curve labeled "Live" corresponds to the total size of live objects, including both compressed and uncompressed. The curve labeled "Compressed" represents the total size of all compressed objects.

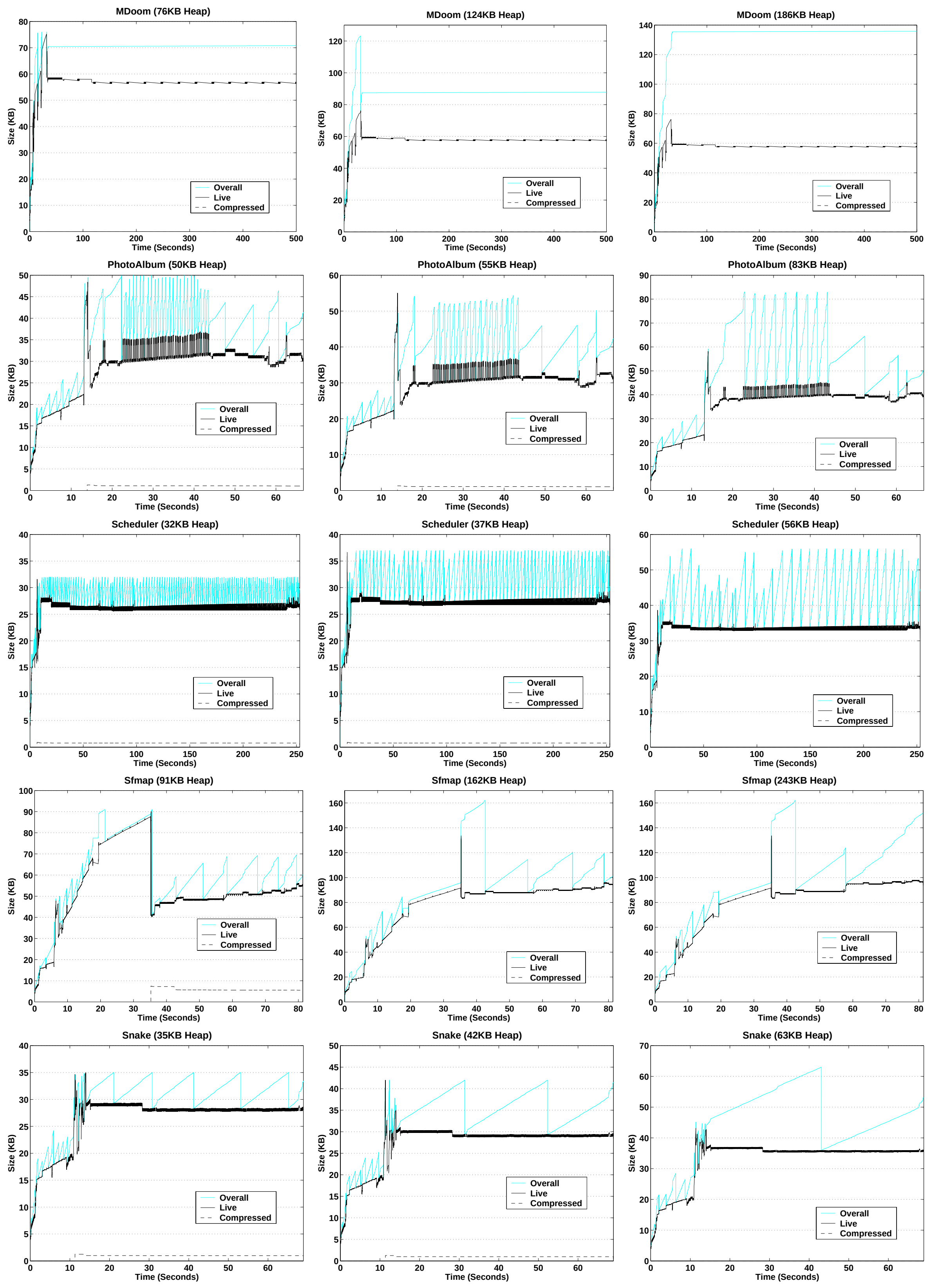


Figure 12: Heap memory usage of MCCL (continued). “Overall” represents the heap usage at a given point during execution, i.e., the total size of live objects (including both compressed and uncompressed objects) and garbage in the heap. The curve labeled “Live” corresponds to the total size of live objects, including both compressed and uncompressed. The curve labeled “Compressed” represents the total size of all compressed objects.

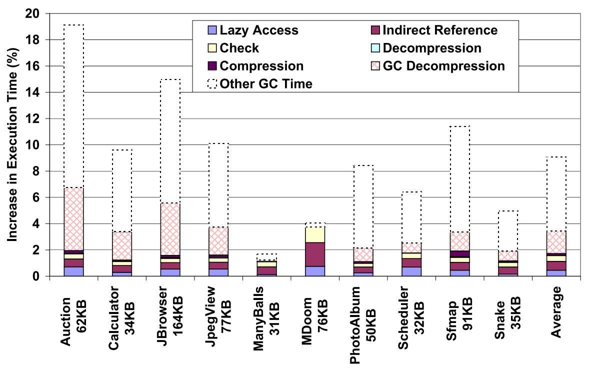


Figure 13: Runtime overheads due to MCCL with the minimum heap size (for MCCL). The values are given as percentages of the MC execution time under an ideal (infinite) heap memory. The numbers next to benchmark names denote the heap sizes used to run the benchmark.

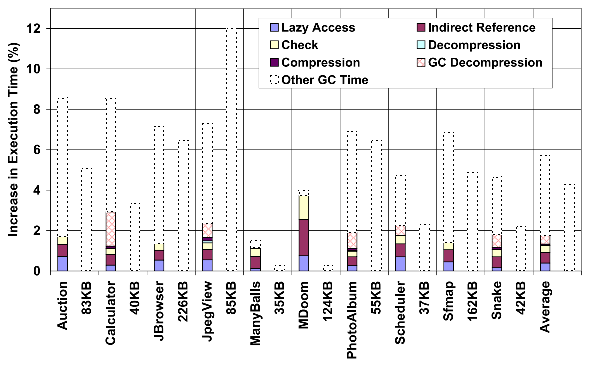


Figure 14: Runtime overheads due to MCCL and MC with the minimum heap size for MC. The values are given as percentages of the MC execution time under an ideal (infinite) heap memory. For each benchmark, the bar on the left is for MCCL, and the bar on the right is for MC. The numbers next to benchmark names denote the heap sizes used to run the benchmark.

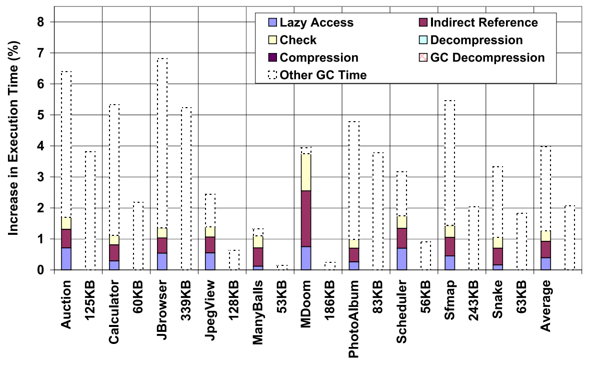


Figure 15: Runtime overheads due to MCCL and MC with 150% of the minimum heap size for MC. The values are given as percentages of the MC execution time under an ideal (infinite) heap memory. For each benchmark, the bar on the left is for MCCL, and the bar on the right is for MC. The numbers next to benchmark names denote the heap sizes used to run the benchmark.

to our compression and lazy allocation based strategy, and occurs even when we use MC with a finite heap). Therefore, the extra overhead introduced by our strategy is very low.

Figures 14 and 15 compare the performance overhead incurred by MCCL and MC when they are used with the same heap size. In Figure 14, for each benchmark, we used the minimum heap size that allows both MC and MCCL to execute without giving an out-of-memory exception (see the third column Table 4). We can see that MCCL is about only 1.5% slower than MC on the average. Note that a performance advantage of MCCL over MC is that it can reduce the number of GC invocations. Consider, for example, JpegView in Figure 14. For this benchmark, with a 85KB heap, MCCL is faster than MC by 4.3%. This is because MCCL reduces the number of GC invocations since the effective heap size is increased. In the graph titled as "JpegView (85KB Heap)" in Figure 11, we observe two invocations of the compressor: at 90th second and at 340th second. These two invocations reduce the application's footprint significantly, which in turn leads to fewer GC invocations. This last observation implies the possibility of using heap compression to reduce the overall garbage collection time even when there is enough heap space (Section 5 evaluates such an aggressive use of object compression). Comparing the graphs in Figure 15 and Figure 14, we observe that, as the heap size increases, the performance degradations (with respect to the ideal heap configuration) due to MC and MCCL are both reduced. This is because, since there is enough space in the heap, the collectors are invoked less frequently.

5. AGGRESSIVE OBJECT COMPRESSION

In our compression based strategies discussed so far, the compression is invoked only when compaction is not successful to provide sufficient free space to accommodate the object to be allocated. In other words, if the compaction is successful in providing sufficient free space, the compression is not activated. Therefore, our approach is oriented towards reducing the impact of compression/decompression on performance. In this section, we investigate the pros and cons of a more aggressive compression strategy. In this strategy, we aggressively compress objects even if just using compaction would allow the application to continue successfully. Our focus is on the MCCL since it is the one that generated the best results so far. Our new collector, denoted MCCL(k), operates with a threshold parameter, k. After the mark phase (of the MCCL), the collector compares the size of the available free space (denoted A) with the size of the object (denoted S) and kH, where H is the size of the entire heap. If

$$A < S \quad \text{or} \quad A < kH,$$

the collector performs compression; otherwise, it compacts the heap without compression. Note that MCCL(0%) is our baseline MCCL strategy discussed so far in the paper. It should also be noticed that the larger the k parameter, the more likely that the compression will be invoked.

In the following, we analyze the impact of this new strategy from the performance and heap memory perspectives. Figure 16 presents the performance impact of k with different heap memory sizes. Each benchmark is run using three different heap sizes. The increases in execution time of MC are also presented for comparison purpose. Obviously, MC results do not change with varying k. On the average, MCCL is slower than MC with the same space by around 2%. We can observe from these results that, when k is increased, with smaller heap sizes, the overall performance overhead of MCCL (which includes all the components as discussed earlier) is decreased. In contrast, when the heap size is larger, increasing the value of k increases the performance overhead. This observation can be explained as follows. Let us assume that two successive garbage collections, GC1 and GC2, are invoked at times t_1 and t_2, respectively. Let us also assume that, right after GC1 is invoked at $t1$, the total size of the objects in the compressed format is c; the original uncompressed size of these objects is a; and the total size of the objects that are decompressed during the interval (t_1, t_2) is d ($d \leq a$). Note that, when an object is decompressed, the compressed version of this object becomes garbage, which remains in the heap until it is collected by the next invocation of the GC. Therefore, if $a > c + d$, the compression invoked during GC1 postpones the invocation of GC2, i.e., if GC1 had not compressed the objects, GC2 would have been invoked earlier than t_2. However, if $a < c + d$, the compression during GC1 actually causes GC2 to be invoked earlier, i.e., if GC1 had not compressed the objects, GC2 would have been invoked later than t_2. Note that a is determined by the total size of the live objects in the heap, and that c is determined by both a and the compression ratio. Neither a nor c is affected by the heap size. However, when the heap size is small, each interval between two successive GC invocations is short (even with compression), which means that the number of objects accessed during (t_1, t_2) is small — which also means that d is small. Therefore, for small heap sizes, compression is more likely to reduce the number of GC invocations. Similarly, for larger heap sizes, the interval (t_1, t_2) is longer and d tends to be larger. As a result, compression is more likely to increase the number of GC invocations. Another factor that also influences the overall performance overhead is the tradeoff between GC cost and compression/decompression cost. A large k value usually results in higher compression/decompression costs. For a small heap size, a large k value tends to reduce the number of GC invocations. If the reduction in the GC costs is larger than the increase in the compression/decompression costs, we are likely to observe a reduction in the overall cost. On the other hand, for a large heap size, a large k value is less likely to reduce the number of GC invocations. Consequently, the overall cost is increased as the value of k is increased.

We next study the heap behavior of MCCL(k) and compare it to MCCL(0%). Figure 17 shows the impact of k on heap behavior with different heap memory sizes. For the minimum heap size that allows each benchmark to run without an out-of-memory exception (the graphs on the left side of Figure 17), we observed that the maximum value of the total size of live objects does not change as the value of k varies. However, for both heap sizes we experimented, we observed that, during most of the execution time, a large k value results in a small total size for the live objects. This is due to the fact that, with a large k value, object compression is performed more frequently, and thus, the heap contains a large number of compressed objects.

6. ELIMINATING OBJECT HANDLES

Our base implementation explained earlier employed object handles mainly because of the difficulty associated with updating reference fields (i.e., the fields that contain references to other objects) of the compressed objects in the compact-compress phase of the collector. Specifically, updating a reference field in a compressed object may cause the size of the object to expand. Since it is not always possible to find the space for the object to expand, our base implementation solved this problem using handles. It should be noted, however, that object handles incur two problems. First, a handle incurs dereferencing overhead whenever the corresponding object is accessed. Second, handle pool occupies space in the heap memory. If the size of the handle pool is small, the application

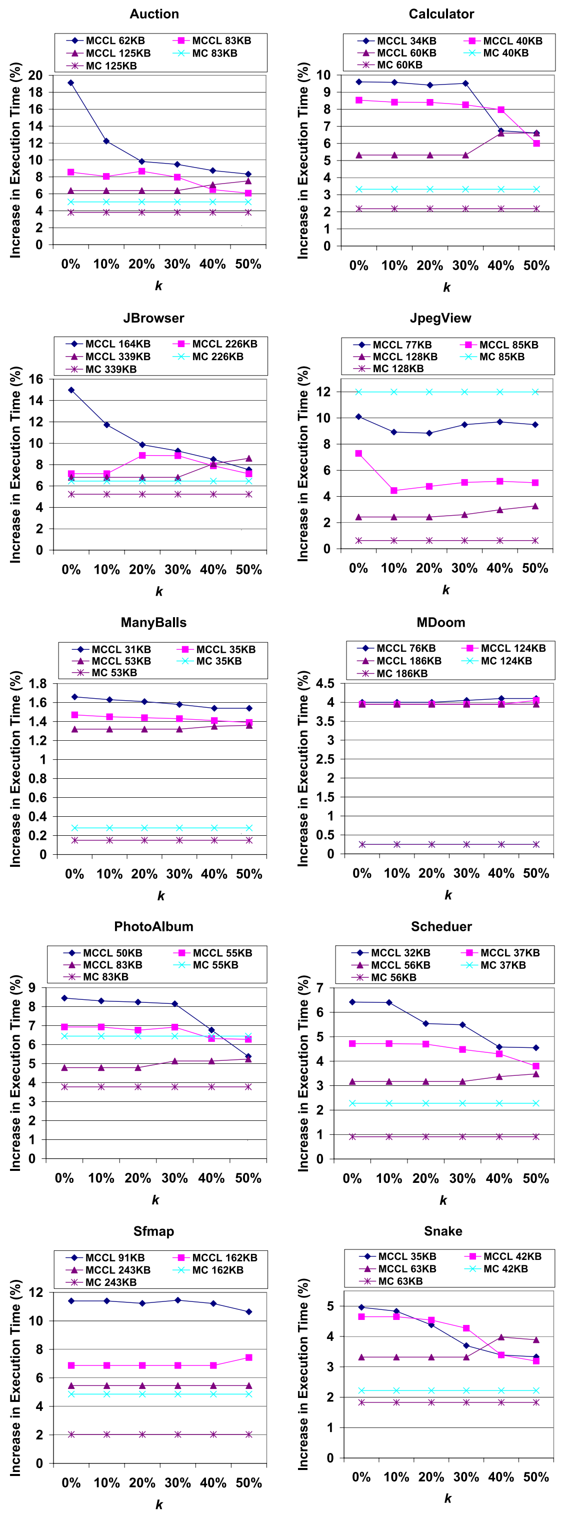


Figure 16: Impact of the threshold parameter k on performance. The influence of varying the k parameter depends on whether a large or a small heap memory is used. The increase in execution time is normalized with respect to the execution time of the MC collector with an infinite heap space. The increase in execution time of MC with finite heap sizes are also presented for comparison purpose (Obviously, MC results do not change with varying k).

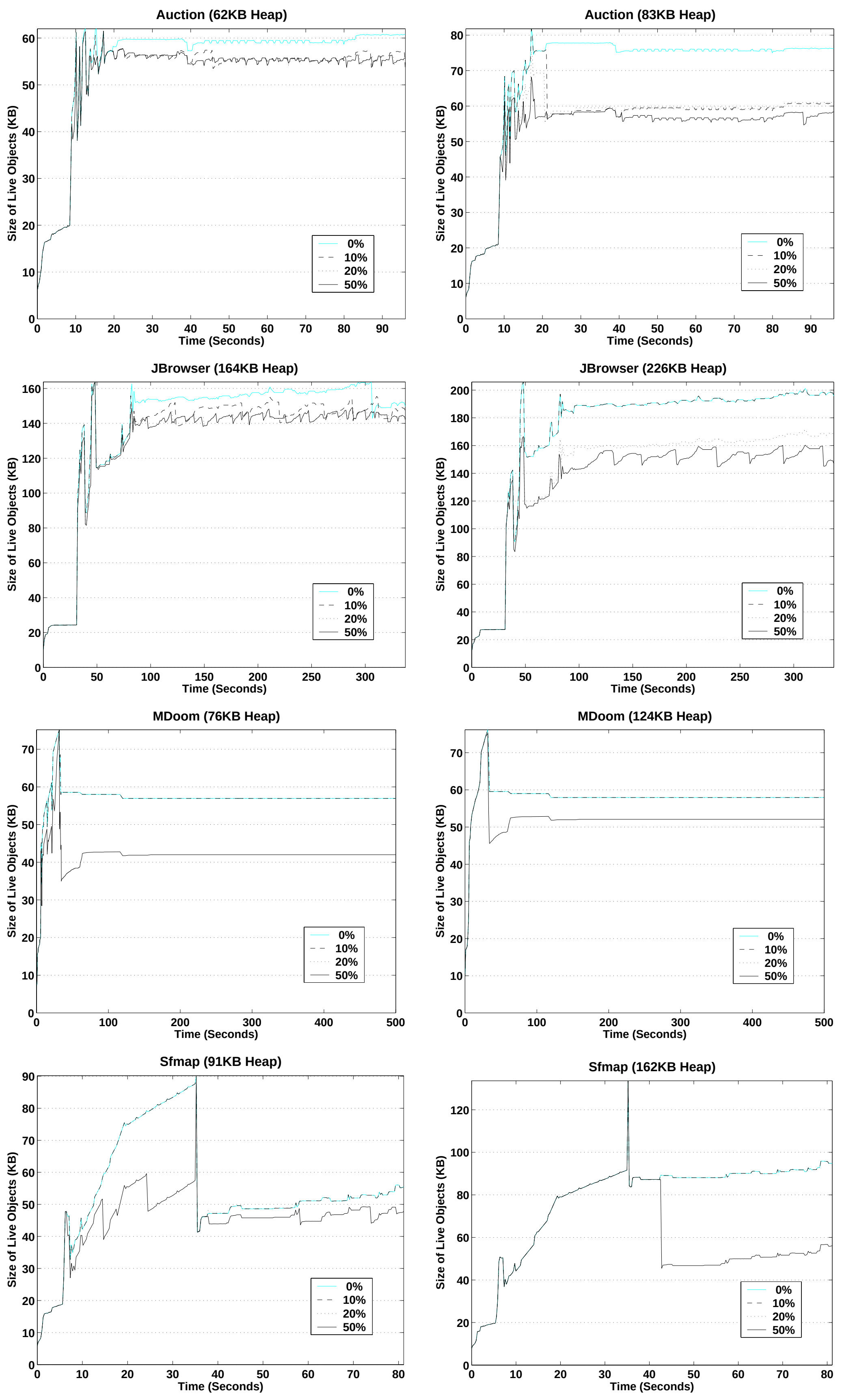


Figure 17: Impact of the threshold parameter k on heap behavior. It can be observed that, in general, increasing the value of the k parameter reduces the heap memory requirement at a given time.

uses up the object handles quickly, which forces frequent GC invocations even if we have available heap space. Consequently, to make the best use of the heap space, we need to fine-tune the size of the handle pool according to the behavior of each application. However, access patterns of different applications may differ from each other dramatically. Further, even the same application can exhibit different heap behavior depending on the user input provided. Therefore, it is very difficult to tune the size of the handle pool successfully. In this section, we present our enhanced implementation that eliminates object handles completely. The cost of doing so is the slight degradation in the compression ratio. However, as will be discussed shortly, we still achieve heap memory savings. Note that the implementation discussed in this section does not use aggressive compression explained in the previous section. It uses our base compression strategy.

Our enhanced implementation divides each object instance into "reference zone" and "non-reference zone" as shown in Figure 18. The reference zone contains only the reference fields, whereas the non-reference zone represents the remaining fields. Our enhanced implementation compresses only the non-reference zone; the fields in the reference zone remain uncompressed. We apply the same technique to arrays as well. The non-reference arrays (that is, the arrays that do not contain references) are compressed as discussed earlier. The reference arrays, on the other hand, are compressed differently, i.e., each bit in the bitmap now corresponds to an element of the array. A 0-bit indicates the corresponding element is null; a 1-bit indicates a reference that is stored in the original format. Keeping the reference fields of each object instance and non-null elements of each reference array in the uncompressed form allows the collector to update the corresponding references whenever an object is moved during the compact-compress phase (without handles).

The process of object decompression in the absence of handles is depicted in Figure 19. After object O1 is decompressed, a forward pointer to the decompressed data is set in the header of the compressed object. When a reference field pointing to the old object is used, our implementation first checks whether the reference needs to be forwarded. If the reference is pointing to an object that has been decompressed, JVM needs to update the reference field to point to the location of the newly decompressed object. In the mark phase of the garbage collection, the collector also checks and updates the reference fields in a similar fashion. It should be noticed that explicitly checking whether a reference has been forwarded for each object access may incur an overhead comparable to using handles. However, we can make use of hardware to forward references transparently. KVM uses 32 bits to represent a reference. To the best of our knowledge, no embedded system today is using more than 2GB memory, which means that the highest-order address bit of each reference is always zero. We use this bit as a flag. If a reference points to a compressed object, this bit is set to 1, otherwise, it remains 0. The system is configured in such way that accessing a non-exist memory location triggers a hardware memory protection exception. We know that the mark-compact garbage collector needs to update each reference in the heap after compaction. Therefore, we modify this procedure so that the compressor also sets the flag bit to 1 for each reference pointing to a compressed object. During the execution of bytecodes, accessing an uncompressed object does not cause any overhead. However, accessing a compressed or forwarded object triggers a memory protection exception. The exception handler then checks if this is an access to a compressed object or to a forwarded object. If it is to a forwarded object, the handler sets the flag of the reference to 0, and then, the virtual machine resumes its execution. If it is to a compressed object and decompression is necessary, the handler decompresses the object.

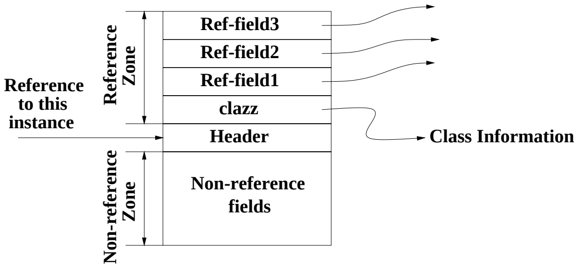


Figure 18: Division of a Java object into reference and non-reference zones. Our enhanced implementation compresses only the non-reference zone; the fields in the reference zone remain uncompressed. We apply the same technique to arrays as well.

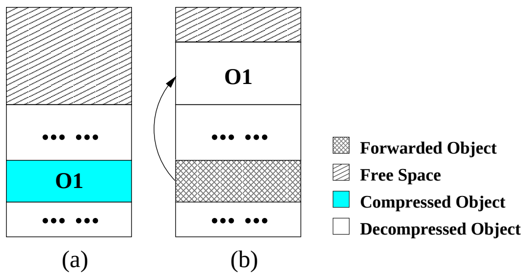


Figure 19: Decompression of object O1 when no handle is used. (a) Before decompression, O1 is in the compressed format. (b) After decompression, a forward reference to the newly decompressed O1 is set in the location that used be to the header of O1.

In the rest of this paper, we denote the MCCL without object handles as "MCCL+" (see Table 1). We can observe from the results in Table 4 that MCCL+ outperforms MCCL for all our benchmarks. This is because of two main reasons. The first is that MCCL+ does not need the handle pool. The second reason is that MCCL+ allows the reference fields of a compressed object to be accessed without decompression. However, for a fair comparison, we need to consider its performance as well. Figure 20 shows the impact of handle elimination on the performance by comparing MCCL+ with MCCL. We see that, for all benchmarks except ManyBalls and Snake, MCCL+ outperforms MCCL. For ManyBalls and Snake, MCCL+ is slower than MCCL, mainly due to frequent accesses to forwarded and compressed objects. On the average, the performance degradations due to MCCL+ and MCCL over the MC with the infinite heap are 5.7% and 9.1%, respectively.

7. DISCUSSION AND FUTURE WORK

Up to this point, we discussed several strategies for reducing the heap memory requirements of embedded Java applications. While our different strategies allow us to explore a large design space, there are still many alternative designs that can potentially be studied. In this section, we discuss several such alternatives and point out the directions for further research.

Selective Compression. In our current implementation, we compress all the live Java objects in the heap. However a more sophisticated object compression strategy can minimize the num-

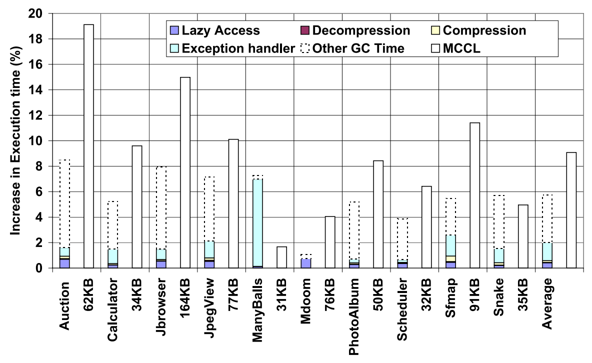


Figure 20: Runtime overheads due to MCCL+. Each benchmark is run with the same heap size as in Figure 13. The overheads are as percentages of the overall execution time of MC with ideal (infinite) heap memory. For each benchmark, the bar on the left is for MCCL+, and the bar on the right is for MCCL (its breakdown into components is as show in Figure 13). The numbers beside the names of each benchmark are the sizes of the heaps used to run the benchmarks.

ber of objects that need to be compressed by considering the size of current allocation request. Specifically, the compressor can (re-)calculate heap memory saving after compressing each object. When the total memory saving from compression plus the size of the free space calculated in the mark phase is larger than the size current allocation request, we can stop the compression process. In addition, it is also possible to rank the objects according to their access frequencies and utilize this information within the compressor to compress only the objects that are not likely to be accessed in the near future.

Independent Compression. In our current implementation, compression is performed during garbage collection. An advantage of this approach is that it is easy to implement. However, it may cause longer pause for garbage collection, which is not desirable in real-time or user-interactive applications. An alternative would be using a dedicated thread to incrementally compress the objects. More specifically, the virtual machine can invoke a compression thread at regular intervals. When the compression thread is scheduled, it selectively compresses a set of objects that will not be used in the near future, and then falls back to sleep. This approach to compression avoids very long pauses due to compression by distributing compressions across the lifetime of the application being executed. The apparent drawbacks include more complicated implementation and extra synchronization overhead.

Using Compression with Generational Collectors. In our implementation, we used compression with a mark-compact collector. However, generational collectors [24, 42] may also employ object compression to reduce the heap memory demands. A generational collector divides the heap into two generations: the young generation and the old generation. All new objects are allocated in the young generation. When the space in the young generation is used up, a local collector is invoked to collect garbage in the young generation. After several local collections, the surviving objects in the young generation are promoted to the old generation. When the space occupied by the old generation reaches a given threshold, a global collector is invoked to collect garbage in both the generations. Compared to the local collector, the global collector is much more expensive. Our compression scheme may be incorporated into the global collector. Specifically, when the virtual machine fails to allocate space for a new object, the global collector is invoked to compress live objects to make space for the new object. It should be noted that, during the peaks of memory demands, the global collector (with compression) may be very costly. Fortunately, as has been mentioned previously, the peaks do not occur frequently and, when they occur, they do not last long. Therefore, in most of the time, the global collector (with compression) may not need to be invoked very frequently.

Hardware-Based Implementation of Compressor and Decompressor. In this work, we employed a software-based implementation for the compressor/decompressor. However, in principle, both the compressor and decompressor may be implemented in hardware as well. There have been several hardware compression schemes proposed in the literature (e.g., [25, 18]). Similarly, the zero removal compression can also be implemented in hardware. Obviously, hardware-based compressor/decompressor is expected to run much faster than the corresponding software-based implementation. Another benefit of the hardware-based implementation is that the compressor and decompressor can work in parallel with the main processor; this can enable us to hide the overhead by overlapping compression/decompression with application execution. For example, when the virtual machine finds that a compressed object is going to be accessed in the near future and that there exist a large amount of heap memory, it can allocate the space to hold the decompressed object and then invoke the hardware decompressor. In this mode of operation, the virtual machine does not need to wait for the decompressor to finish its work. Instead, it can continue with the application execution as long as the object in question is not accessed immediately. When the virtual machine really needs to access the object, it is very likely that the decompressed object will be ready to be used. In other words, using a hardware-based compressor/decompressor, one can implement a “pre-decompression” scheme that can significantly reduce the overhead associated with accessing the objects.

8. RELATED WORK

The work presented in this paper is related to the prior studies in the areas of garbage collection, memory compression, and memory footprint reduction in object-oriented programs. In this section, we present a discussion of the prior work in these areas.

Garbage Collection Strategies. Garbage collection has been an active area of research for the past years [24, 42]. The studies in this area can be broadly classified into two categories: improving the performance (or energy consumption, predictability of pause time, etc.) of the collector itself and using the GC to improve the performance (or, energy consumption) of the mutator. Since it is not possible to cover all the prior work here, in the following, we discuss some representative studies from these two categories.

The examples in the former category mentioned above include [12, 13, 10]. Blackburn et al. [12] present a framework —Beltway— that significantly generalizes existing copying collectors. More importantly, the Beltway framework enables design and implementation of new collectors that are robust to variations in heap size and improve total execution time. Their observation is that a garbage collection scheme based on Beltway framework is faster than the best generational copying collectors by up to 40%, and on the average by 5% to 10%, for small to moderate heap sizes. [13] uses pre-tenuring advice to improve the performance of generational and Old First collectors. The novelty of this work is that their pre-tenuring advice is neutral to the garbage collector algorithm and configuration. [10] is an example that focuses on the predictability of the garbage collection pause time. The authors present an algorithm that achieves not only stable pause times at real-time resolution, but also highly predictable processor utilization rates for the mutator.

The examples in the latter category include [38, 15, 14]. Shuf et al. [38] present techniques that create and preserve locality of Java applications at both allocation and garbage collection times. Their locality-based traversal technique reduces garbage collection time by up to 20% (10% on the average) and improves performance by up to 14% (6% on the average). Combining with their locality-aware allocation technique, they improve the performance of the applications by up to 22% (10% on the average). Chilimbi et al. [15] use generational garbage collector to implement cache-conscious data placement. Their results indicate that the proposed technique reduces cache miss rates by 21%–42% and improves program performance by 14%–37% over the Cheney's copying algorithm. Chen et al. [14] use the GC to reduce the energy consumption of embedded Java applications. Their scheme assumes a banked memory architecture and demonstrates that increasing the GC frequency helps increase the number of banks that can be placed into a low power operating mode.

Escape analysis (e.g., [16]) provides another solution that reduces the burden of garbage collection. Specifically, an escape analyzer determines if an object can be allocated in the stack and if an object is accessed by only a single thread. In this way, the objects allocated in the heap can be collected automatically when the corresponding method returns and the synchronizations on the objects that are accessed by only a single thread can be safely removed. Choi et al. [16] show that, for their benchmarks, up to 70% of objects can be allocated in the stack and 11% to 92% of lock operations can be safely removed. Putting all together, they observe performance improvements ranging from 2% to 23%.

Another research area related to garbage collection is characterization of memory behavior. The results from the studies in this area may be used to guide the optimization of GCs. Dieckmann et al. [19] analyze the memory allocation behavior of the six Java benchmarks from the SPECJVM98 suite. Shuf et al. [39] characterize the inherent memory (e.g., TLB and cache) behavior of Java workloads. Hirzel et al. [23] explore the connectivity of Java objects in the heap. Their observation is that connectivity correlates strongly with object life-times and death-times.

Memory Compression. Compression has been employed in embedded systems to reduce the cost, space or energy consumption of memory. Lekatsas et al. [26, 27] propose instruction code compression as an efficient method for reducing power consumption on an embedded system. Their experimental results indicate that their schemes bring energy savings between 22% and 82%. Chen et al. [22] use hardware memory compression to reduce the leakage energy consumption of the system memory in an embedded Java environment. In their work, they achieve energy savings through compression of Java binary code and the pre-loaded Java class library. Clausen et al. [17] compress Java bytecodes using factorization of common sequences. Their scheme targets low-end embedded systems and reduces the memory space occupied by Java bytecodes to 85% of the original size on the average, with a slight execution time penalty. Pugh [33] develops a format for compressing Java classes. The Java classes compressed using this format are typically 1/2 to 1/5 of the size of the corresponding compressed jar files (and 1/4 to 1/10 the size of the original class files).

Memory compression may also be used in high-performance systems to increase the effective size of the memory and reduce the I/O cost due to page swapping. Franaszek et al. [21] develop a set of algorithms and data structures for compressed-memory machines. Their algorithms are implemented in IBM Memory Expansion Technology (MXT). For typical systems, their techniques yield a factor of 2 expansion in effective memory size. Rizzo [34] presents a very fast algorithm for RAM compression. The author suggests that using this compression algorithm can lead to both memory savings and performance improvements in servicing page faults.

Footprint Reduction in Object-Oriented Systems. Eckel et al. [20] observe that current C++ compilers generate a significant number of fields that are used internally. Such internal fields are considered as memory overheads of the object layout. They also find that, by using inlining and bidirectional object optimization techniques, this overhead can be reduced by nearly 50% on the average. Bacon et al. [11] present an implementation of the Java Object Model that is efficient from both the space and time angles. Their implementation achieves an average memory saving of 7%. Shaham et al. [35] present a heap-profiling tool for exploring the potential for space savings in Java applications. The output of the tool is used to direct the application source code rewriting in a way that allows more timely garbage collection of objects, thereby saving space. By manually rewriting the benchmarks' source code, they report 18% space savings, on the average, for the applications in the SPECJVM98 benchmark suite. Shuf et al. [37] distinguish the types of Java objects according to the number of instantiated objects of each type. Their approach improves pause times, eliminates unnecessary write barriers, and reduces garbage collection time by up to 15% (compared to the corresponding generational collector). Their scheme also reduces the heap space requirements of applications by up to 10%. Shaylor [36] implements a Java JIT compiler for memory constrained low-power devices. His JIT speeds up execution by a factor of 5.7 to 10.7, and its implementation requires only 60KB of the ARM machine code. He also uses a quick and simple management of the JIT code buffer to minimize the memory required for storing the compiled code.

In addition to the Sun Microsystem's J2ME technologies [4, 3], there exist many other implementations of Java or non-Java virtual machines with small memory footprint in mind. For example, McDowell et al. [32] present a Java environment that supports the complete Java language and all the core Java packages except AWT using as little as 1MB of RAM, without any additional ROM (except a small boot ROM) or disk. TinyVM [8] is an open source Java platform for the Lego Mindstorms RCX microcontroller. TinyVM's footprint is about 10 KB in the RCX. Additionally, program class files are compacted considerably before they are downloaded to the RCX. A small program can access around a 16 KB of RAM. An example non-Java small-footprint virtual machine is Maté [28], a tiny communication-centric virtual machine designed for sensor networks. The high-level interface of Maté allows complex programs to be very short (under 100 bytes). Maté and all of its subcomponents can accommodate in 1KB RAM (for data) and 16KB ROM (for instructions).

Our Work. The work described in this paper is different from all these studies. Our approach to reducing the heap footprint of Java applications is based on object compression. To the best of our knowledge, none of the previous papers in this area explored this alternative. However, it should also be noted that our approach can be used in conjunction with some of prior studies (e.g., [11, 35]) to reduce memory footprint even further. In addition to allowing an application to execute under severe heap memory constraints, our approach also reduces the percentage of memory used at any given point in execution. The memory saved can be used for saving power, and in fact, our strategy can be used as a part of a larger framework which also includes [14] and [22].

9. CONCLUDING REMARKS

Reducing memory demands of Java applications is critical for embedded systems as these systems operate under memory constraints. An effort in this direction, if successful, can increase the number of Java applications that can execute in systems with low memory budget. In this paper, we present a set of heap management strategies for reducing memory footprint. Our major conclusions can be summarized as follows:

- Our compression-based garbage collection strategy, MCC, reduces the minimum heap size required to run Java applications by 10.8%, on the average, over the MC collector. The corresponding memory saving over the MS collector is 40% across all ten benchmarks. These results are obtained by using object compression when it is absolutely necessary to continue executing the application without an out-of-memory exception.

- Our lazy allocation and object break-down technique and the garbage collection strategy based on them, MCL, also reduce the required heap sizes significantly (9.5% on the average). In addition, combining MCC and MCL under the integrated strategy MCCL increases memory savings further (21% on the average).

- In addition to reducing the minimum heap sizes to execute applications, our garbage collectors also reduce the total size of the live objects that need to be kept in the heap. This can be exploited, among other things, for reducing energy consumption of the memory system or increasing the concurrency in a multiprogrammed environment.

- The performance degradation (with respect to the MC collector that uses an infinite heap memory) caused by MCCL is affected by the actual heap size that is available to the application. Generally, the larger the heap size, the smaller the degradation. Using the minimum heap size that allows the application to run without out-of-memory exception, the average performance degradation of MCCL (over MC with infinite heap) was found to be 9.1%. However, most of this overhead (5.6%) is due to the garbage collection activity itself, which should occur with any collector with finite heap. In fact, our results show that MCCL is less than 2% slower than MC with the same heap size.

- Our experiments with a more aggressive compression strategy (which compresses objects even if is is not strictly necessary to do so) indicate that such a strategy improves the performance of our benchmarks over the baseline MCCL by up to 9.2% (5.5% on the average) with the minimum heap size.

- Our enhanced implementation that does not use object handles, called MCCL+, improves both memory behavior and performance degradation of the base MCCL. The main reasons for memory savings are that MCCL+ does not need handle pool (which occupies a significant amount of heap space), and that the reference fields can be accessed without decompressing the corresponding object. The main reasons for performance improvements are that, in MCCL+, objects can be accessed directly without incurring dereferencing, and that MCCL+ eliminates the overheads due to the handle pool maintenance.

- Our experience with compression, lazy allocation, and object break-down suggests that these strategies can also be used in conjunction with different base collectors (e.g., generational), virtual machines, and compression algorithms.

10. REFERENCES

[1] 8/16 meg PalmPilot upgrades - prices. http://www.palmpilotupgrade.com/prices.html.

[2] Casio: CdmaOne C452CA. http://www.javamobiles.com/casio/c452ca/.

[3] Connected device configuration (CDC) specification. http://java.sun.com/j2me/.

[4] Connected limited device configuration (CLDC) specification. http://java.sun.com/j2me/.

[5] Java2 platform micro edition (J2ME) technology for creating mobile devices (white paper). http://java.sun.com/products/cldc/wp/KVMwp.pdf.

[6] Mobile information device profile (MIDP) specification. http://java.sun.com/j2me/.

[7] Palm product - Palm handheld comparison chart. http://www.palm.com/products/family.epl.

[8] TinyVM. http://tinyvm.sourceforge.net/index.html.

[9] US mobile devices to 2006: A land of opportunity. Based on an extensive research program conducted by Datamonitor on mobile device markets in the US.

[10] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 285–298, New Orleans, Lousiana, USA, Jan. 2003.

[11] D. F. Bacon, S. J. Fink, and D. Grove. Space- and time-efficient implementation of the Java object model. In *the 16th European Conference on Object-Oriented Programming (ECOOP'02)*, University of Málaga, Spain, June 2002.

[12] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: getting around garbage collection gridlock. In *the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, pages 153–164, Berlin, Germany, June 2002.

[13] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for Java. In *the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'01)*, pages 342–352, Tampa Bay, FL, USA, 2001.

[14] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection in an embedded Java environment. In *the 8th International Symposium on High-Performance Computer Architecture (HPCA'02)*, Cambridge, MA, USA, Feb. 2002.

[15] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *the 1st International Symposium on Memory Management (ISMM'98)*, pages 37–48, Vancouver, British Columbia, Canada, 1998.

[16] J. D. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, Denver, CO, USA, Nov. 1999.

[17] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems (TOPLAS'00)*, 22(3):471–489, May 2000.

[18] D. J. Craft. A fast hardware data compression algorithm and some algorithmic extensions. *IBM Journal of Research and Development*, 42(6), 1998.

[19] S. Dieckmann and U. Holzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, Lisbon, Portugal, June 1999.

[20] N. Eckel and J. Gil. Empirical study of object-layout strategies and optimization techniques. In *the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, Sophia Antipolis and Cannes, France, June 2000.

[21] P. A. Franaszek, P. Heidelberger, D. E. Poff, and J. T. Robison. Algorithms and data structures for compressed-memory machines. *IBM Journal of Research and Development*, 45(2):245–258, Mar. 2001.

[22] G.Chen, M.Kandemir, N.Vijaykrishnan, and W.Wolf. Energy savings through compression in embedded Java environments. In *the 10th International Symposium on Hardware/Software Codesign (CODES'02)*, Colorado, USA, May 2002.

[23] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects (ismm'02). In *the 3rd International Symposium on Memory Management*, pages 36–49, Berlin, Germany, 2002.

[24] R. Jones. *Garbage Collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Ltd, 1999.

[25] M. Kjelso, M. Gooch, , and S. Jones. Performance evaluation of computer architectures with main memory data compression. *Elsevier Science, Journal of Systems Architecture*, 45:571–590, 1999.

[26] H. Lekatsas, J. Henkal, and W. Wolf. Code compression for low power embedded system design. In *the 37th Conference on Design Automation (DAC'00)*, pages 294–299. ACM Press, 2000.

[27] H. Lekatsas and W. Wolf. SAMC: a code compression algorithm for embedded processors. *IEEE Transactions on CAD*, 18(12):1689–1701, Dec. 1999.

[28] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, San Jose, CA, USA, Oct. 2002.

[29] D. Lidsky and J. Rabaey. Low-power design of memory intensive functions case study: Vector quantization. In *1994 IEEE Workshop on VLSI Signal Processing*, pages 26–28, La Jolla, CA, USA, Oct. 1994.

[30] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification Second Edition*. Addison-Wesley Pub Co, 1999.

[31] S. McAteer. Java will be the dominant handset platform. http://www.microjava.com/articles/perspective/zelos.

[32] C. E. McDowell, B. R. Montague, M. R. Allen, E. A. Baldwin, and M. E. Montoreano. Javacam: Trimming Java down to size. *IEEE Internet Computing*, 2(3), May/June 1998.

[33] W. Pugh. Compressing Java class files. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 247–258, 1999.

[34] L. Rizzo. A very fast algorithm for RAM compression. *ACM SIGOPS Operating Systems Review*, 31(2):36–45, Apr. 1997.

[35] R. Shaham, E. K. Kolodner, and S. Sagiv. Heap profiling for space-efficient Java. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 104–113, 2001.

[36] N. Shaylor. A just-in-time compiler for memory constrained low-power devices. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'02)*, San Francisco, CA, USA, Aug. 2002.

[37] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *Symposium on Principles of Programming Languages (POPL'02)*, pages 295–306, 2002.

[38] Y. Shuf, M. Gupta, H. Franke, A. Appel, and J. P. Singh. Creating and preserving locality of java applications at allocation and garbage collection times. In *the 2002 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*, Seattle, WA, USA, Nov. 2002.

[39] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *SIGMETRICS/Performance*, pages 194–205, 2001.

[40] C. van Reeuwijk and H. J. Sips. Adding tuples to Java: a study in lightweight data structures. In *Joint ACM Java Grande – ISCOPE 2002 Conference*, pages 185–191, Seattle, WA, USA, Nov. 2002.

[41] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of Java applications from the memory perspective. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, Monterey, CA, USA, Apr. 2001.

[42] P. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, Saint-Malo, France, 1992. Springer-Verlag.

[43] Y. Xie, W. Wolf, and H. Lekatsas. A code compression architecture for VLIW processors. In *the 34th Annual International Symposium on Microarchitecture (MICRO'01)*, pages 66–75, Austin, TX, USA, 2001.