Protected Heap Sharing for Memory-Constrained Java Environments

Yoonseo Choi and Hwansoo Han
Division of Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, 305-701, Republic of Korea
yschoi@arcs.kaist.ac.kr, hshan@cs.kaist.ac.kr

ABSTRACT

Multitasking is one of capabilities we often want to have in memory-constrained embedded systems. To support multiple address spaces within a small physical memory, a simple memory management frequently encounters the lack of available memory. Our paper presents an efficient heap memory management scheme that reduces memory footprints by adaptively sharing heaps among multiple tasks in JVM environments. We modified KVM from Sun Microsystems so that Java applications acquire or release heaps in a shared pool on an as-needed basis. To protect address spaces among tasks in the absence of virtual memory capabilities, we use memory protection units (MPUs) by incorporating them into our heap sharing scheme. Our experiments with J2ME MIDP applications show significant reductions by 33% on average, ranging from 6% to 50% in memory usage over the execution. The overheads of our scheme in garbage collection are kept low. The execution times in our scheme increase only by 0.2% on average.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management*; D.3.4 [**Programming Languages**]: Processors—*memory management(garbage collection)*; D.3.4 [**Operating systems**]: Storage Management—*allocation/ deallocation strategies, garbage collection, virtual memory*

General Terms

Performance, Design

Keywords

dynamic memory management, memory protection unit, garbage collection, heap sharing

1. INTRODUCTION

Multitasking is a growing trend in many small embedded systems. In spite of advances in processors and operating systems, many embedded systems are suffering from the limited size of physical memory and the lack of virtual memory capabilities, which impedes both the flexible execution and rapid development of application software. Without virtual memory, all application programs should run within the given physical memory area. Since compilers and linkers usually generate code on a contiguous address space, applications are provided with the same-sized contiguous physical memory space to minimize modifications to the code generation tools (refer to Figure 1 (a)). Evenly distributing memory space to all applications, however, may result in inefficient usage of memory in multitasking. One program can be aborted due to the shortage of memory while another program runs only partially using allocated memory. We often observe that the amount of live objects changes over the execution and the peak of memory usage lasts only for a short period. Considering the changes in memory usage, we can build a more effective memory management for multitasking by giving surplus memory of one task to another memory-hungry task. Since the amount of memory used in the heap segment is dominant over those in code, data and stack segments, our study focuses on reducing the amount of memory spent in the heap. To realize our focused goal, we propose a method for sharing heap memory among multiple applications. The target systems for our study are the small embedded devices that lack the virtual memory capability. Figure 1 (b) depicts the conceptual design of memory usage. The operating system provides a contiguous address area for code, data and stack, while it provides heap with fragmented areas out of a shared pool. Note that a task can have more than one non-contiguous heap areas.

Allocating heap in a shared pool without the virtual memory capability incurs several problems such as: (1) how a task seamlessly manages multiple fragmented heap areas, (2) how large each heap area should be, (3) when heap areas should be given to and reclaimed from applications, and (4) how we protect memory address space including all heap areas from other programs' undesirable accesses. Our proposed scheme fully resolves all these problems. We modified KVM [8] from Sun Microsystems to manage the fragmented heap areas. Our modified KVM acquires heap areas and releases the heap areas with proper granularity as the heap memory usage changes. We protect address space by incorporating the protection mechanism of the memory protection unit (MPU) from ARM processors into our heap sharing scheme. The MPU can protect multiple memory segments at lower costs than the memory management unit (MMU), and is adequate for multitasking in the

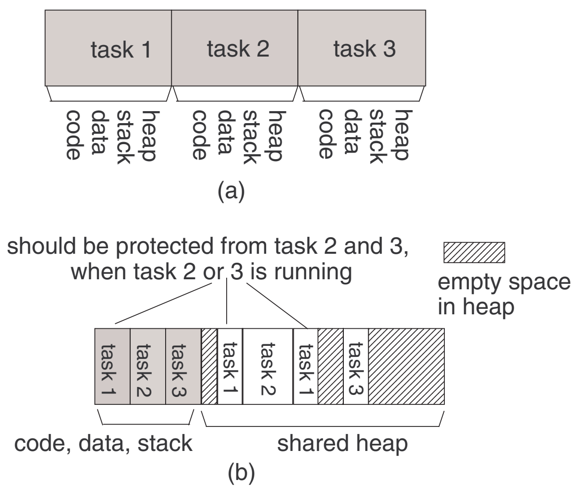

Figure 1: (a) simple memory assignment, (b) heap sharing among multiple tasks

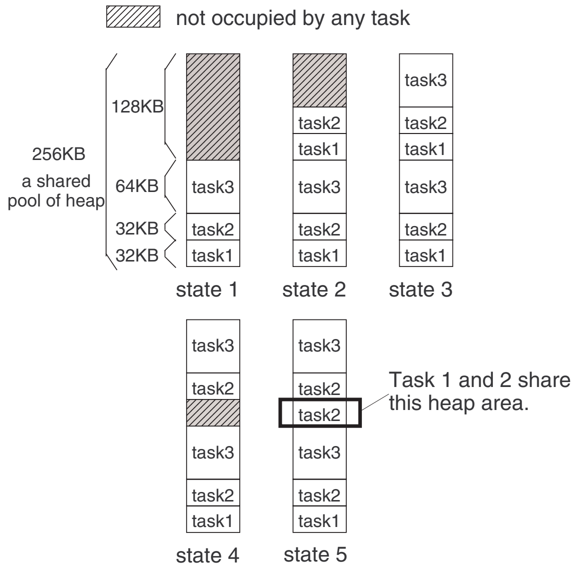

Figure 2: Acquisitions and releases of heap areas out of the shared pool for three simultaneous tasks.

embedded systems with no virtual memory support. The MPU associates a memory area to a protection domain which imposes an access criterion to that area. Most modern processors provide at least two modes of operation: user mode and privileged mode. By comparing the access permissions of associated domains with current operation modes of processors, the MPU examines whether the accesses to the corresponding memory areas are legal.

Sharing heaps and protecting fragmented heaps, our proposed memory scheme achieves the following goals.

- It reduces a considerable amount of memory footprint by smartly exploiting the phases in memory usage pattern of application programs.
- It protects all fragmented address spaces of a program using the MPU.
- It shows almost no increase in execution time by carefully triggering garbage collections and live object compactions.

The remainder of this paper is organized as follows. We provide an overview of our method for sharing a heap among multiple applications in Section 2. We describe the design and implementation of our memory management scheme in Section 3. We experimentally examine the effectiveness of our scheme in Section 4. Finally we discuss related work and conclude this paper.

2. HEAP SHARING AND PROTECTION

2.1 Allocating Heaps from A Shared Pool

Tasks in multitasking environment often have different memory requirements in size. Suppose there are three tasks to run together on a device equipped with 256 KB heap memory and each of them uses up to 64 KB, 96 KB, 128 KB heap memory, respectively. To focus our discussion on heap we assume the memory required for code, data, and stack is allocated outside the 256 KB heap memory. Using a simple area assignment, tasks need to secure contiguous memory areas of at least 64 KB, 96 KB, and 128 KB, respectively. Since those three areas cannot fit in 256 KB heap memory ($256 < 64+96+128$), one of the tasks cannot start due to the shortage of memory if we simultaneously launch all three tasks.

On the other hand, if tasks begin with only the required amount of memory for their starts and adaptively acquire and release memory on an as-needed basis, we may be able to launch all three tasks and execute them to the end within the 256KB memory. Figure 2 illustrates the described situation. Task 1, task 2 and task 3 start with 32 KB, 32 KB, and 64 KB heaps, respectively at state 1. Each of them acquires another heap area based on its need at state 2 and 3. Task 1 releases one of its heap areas at state 4, which is used by task 2 later at state 5. Note that a fragmented heap area is shared between two tasks. The heap fragment marked with a bold rectangle in state 5 is once occupied by task 1 and later by task 2. In this way of sharing, task 1 gives its surplus heap memory to task 2.

2.2 Address Space Protection for Individual Tasks

Fully utilizing the facility of MPUs, we protect fragmented address areas of one task. MPUs provide an effective address space protection even though they do not support virtual memory systems as the MMUs do. MPUs associate an address area with a region which grants an access right such as no-access, read-only, and read-write to that address area. Several ARM cores such as ARM740T, ARM940T, ARM946E-S and ARM1026EJ-S are equipped with the MPUs [3]. There are slight variations in the MPUs among ARM processors. ARM740T, ARM946E-S, and ARM1026EJ-S have eight unified instruction and data regions. ARM940T has sixteen regions: eight regions for instruction segments and additional eight regions for data segments. Since we focus on heaps which are more dynamic and larger in size than code, data, and stack, we take only eight data regions into our consideration for address space protection.

Each region in the MPUs is identified and prioritized by its number. The attributes of regions, such as the starting addresses, lengthes, access rights, and caching polices, are configured by the operating system. All these configurations can be modified by updating special registers (CP15 registers) in ARM processors. When a higher priority (i.e. larger number) region overlaps with a lower priority region, accesses to the overlapped region are controlled by

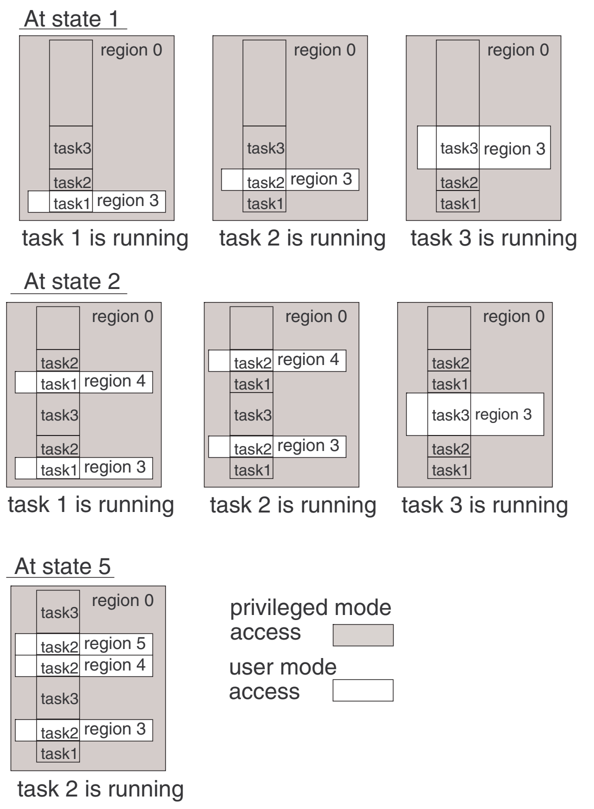

Figure 3: The protection of memory address areas using MPU regions.

the access permission of the higher priority region. Figure 3 illustrates how to use MPU regions for protecting address areas of each application program which is previously shown in Figure 2. Shaded regions have read-write permission for privileged accesses and no-access permission for user accesses, while white regions have read-write permission for privileged accesses and user accesses. Since the priority of region 3 is higher than region 0, tasks can access only the memory areas covered by region 3 when they are actively running (state 1). The memory areas of the other non-active tasks are protected by region 0. We assume the operating system updates the attributes of region 3 during context switches. By updating the configuration of region 3 with proper memory areas of a running task, we can effectively protect the address space of each task. To protect later obtained heap areas of task 1 and 2 at state 2, we use another MPU region (region 4) which is also prioritized higher than region 0 and has the read-write permission for privileged and user modes. We use up to three MPU regions at state 5 to protect all heap areas of an active task.

Restrictions in MPU regions Recall that MPU provides eight regions. An application program can occupy up to four separate regions since usually four out of eight regions are used for other purposes such as background memory, kernel memory, shared memory, and memory mapped area of peripheral devices. In addition, the size of each region should be a power of two between 4 KB and 4 GB. The example shown in Figure 3 also conforms to the limitations imposed by MPUs. All regions are power of twos in size and each task uses less than or equal to four regions.

The role of the operating system When a task requests the operating system to allocate more heap memory, the *global memory manager* in the system checks whether there is a remaining MPU region. If any, it associates that remaining region to a free block with a proper size. The priority of the region is given at higher than 0, since region 0 covers the background region for protection. The access permission of the region is read-write for privileged mode and user mode, since heap will contain data storages that can be read from and written to by a user task. The global memory manager sets the attributes of the region by writing to special registers. For example, the global memory manager introduces region 4 at state 2 in Figure 3 by setting the attributes of region 4 to protect the newly obtained heap area of task 1 and 2. The global memory manager also keeps the records of unused regions and maintains the list of free memory blocks in a shared heap. Since the regions and their access permissions used by tasks are different, the operating system stores those information in the corresponding process control blocks. Whenever context switches occur, the operating system stores the settings of MPU regions for suspended tasks and updates the settings of MPU regions for the tasks to schedule.

When a task requests to release a heap area, the global memory manager sets the associated region back to unused state and updates the information on unused regions and free memory blocks.

The role of application programs An application program should be able to recognize whether it needs more memory or it is ready to relinquish some of its memory back to the system. Knowing when it needs more memory is easy. When applications fail to allocate memory within given the heap fragments, they can request more heap memory. To be able to return surplus memory, applications need to free up non-live data. If a region contains only the non-live data, they can return that region to the system. Compaction can be used to find an available free space before applications request more heap area. Applications also use compaction to find surplus space to release. We will describe the roles of applications in the next section in the context of Java applications and Java virtual machines. In addition, applications should be able to manage non-contiguous address space resulting from the acquiring and releasing of memory fragments in and out of a shared pool.

Many Java programs require a significant portion of their address space for heap area compared to the areas for code, data, and stack. In addition, Java applications are becoming prevalent in small embedded devices. Our paper focuses mainly on Java applications for embedded systems. Recall that Java's protection mechanism such as no pointers and range check enables multiple java applications to run together inside a single JVM sharing a single large heap area. However, when native function calls are allowed, malicious applications can access to other applications' heap objects through pointer arithmetics ruining the protection of the memory area of each application. On the other hand, our proposed approach can successfully protect the memory areas of all applications even in the existence of native function calls by separating the heap area of one application from those of other applications and protecting address spaces through MPU regions. By modifying memory allocators and garbage collectors in Java virtual machines, we can make applications work with our heap sharing system.

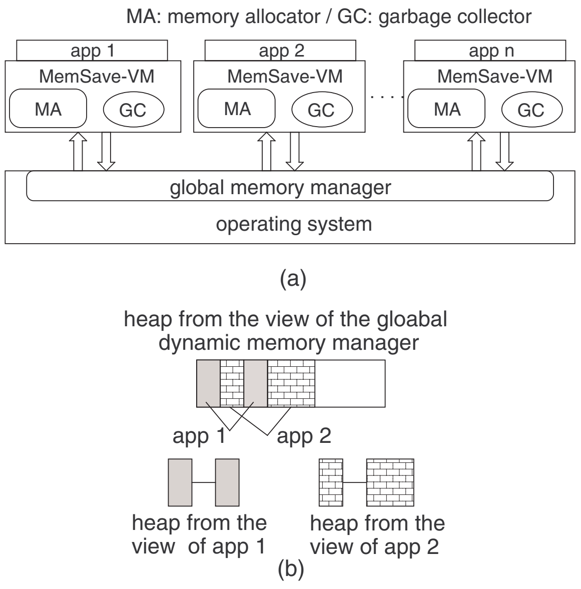
MA: memory allocator / GC: garbage collector
app 1
app 2
app n
MemSave-VM
MA
GC
global memory manager
operating system
(a)
heap from the view of the gloabal dynamic memory manager
app 1
app 2
heap from the view of app 1
heap from the view of app 2
(b)

Figure 4: An overview of the design: (a) the roles of the operating system and an application program, and (b) the view on heap from each side.

3. MEMORY SAVING JAVA VIRTUAL MACHINE (MEMSAVE-VM)

Figure 4 (a) provides an overview on how the operating system and multiple applications interact to share a common pool of heap. Whenever the Java virtual machine(JVM) for an application needs more space for its heap, its own memory allocator requests the global memory manager to give a heap area. On the other hand, the garbage collector in the JVM relinquishes its surplus heap areas back to the global memory manager. Responding to the requests from applications, the global memory manager in the operating system is in charge of allocating and reclaiming heap areas. Heaps from the view of the global memory manager and applications are respectively shown in Figure 4 (b). Sharing heap among multiple tasks is achieved by passing a fragmented heap area from a task to another via the global memory manager based on the requests of tasks to acquire and release.

Our JVM, called MemSave-VM, is aware that its heap is fragmented and identifies each fragmented heap area as a *sub-heap*. MemSave-VM is modified from KVM of Sun Microsystems. KVM is a compact, portable Java virtual machine that has been designed specifically for small, resource-constrained devices [9]. Note that how the heap layout of MemSave-VM is different from that of the original KVM in Figure 5. KVM manages a contiguous, fixed size heap. Meanwhile, the heap of MemSave-VM is comprised of one or more sub-heaps which are non-contiguous in address, and the size of the heap changes as MemSave-VM acquires and releases sub-heaps. In the original KVM only the boundary between the group of permanent objects and the group of dynamic objects moves towards the start of the heap when garbage collectors compact dynamic objects to expand the space for permanent object allocations. While dynamic objects are garbage-collected, the permanent objects that contain class related information such as bytecodes and constant pools, are not garbage-collected. They stay alive until the end of the execution. Unlike KVM, MemSave-VM can avoid the compaction of dynamic objects only to get the space for more permanent objects. By putting permanent objects into separate sub-heaps from dynamic objects MemSave-VM eliminates the intervention between two types of objects. Furthermore, this segregation is generally more effective on memory space saving since it allows you to acquire and release sub-heaps in a more flexible way. On the contrary, if permanent objects and dynamic objects reside together in a sub-heap, that sub-heap can never be released since permanent objects are never garbage-collected.

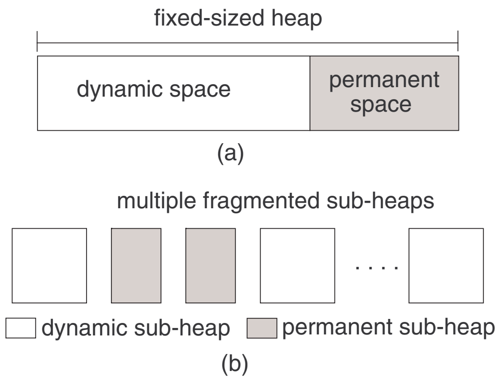

Figure 5: The layout of heap in KVM and MemSave-VM: (a) KVM with mark-sweep compact GC, and (b) MemSave-VM.

The memory allocator in our MemSave-VM cooperates with the global memory manager during the allocation of an object according to the following procedure.

(1) The allocator searches the list of free blocks to find a free block large enough to hold the object.

(2) If it finds a free block, it simply returns the free block. Otherwise, it invokes a garbage collection, and then retries the search on the revised list of free blocks.

(3) If the allocator still fails to find an adequate free block, it requests more heap memory from the global memory manager in operating system. Once the allocator acquires a new heap area, it maps the heap area to one of its sub-heaps and also enlists the heap area into the list of free blocks. The allocator now can find a free block for the object.

(4) If the allocator is not provided with an additional heap area from the global memory manager, the application program is aborted.

Since the original KVM works only with the given heap at the beginning of the execution, the steps (3) and (4) are not its options.

Our main concern is to design a memory allocator and a garbage collector capable of managing sub-heaps as discussed above. Since a sub-heap is the unit of an acquisition and a release, we discuss in the following sections (1) how large a sub-heap size should be, (2) how MemSave-VM maintains the list of free blocks of heap, and (3) through what procedure Memsave-VM releases sub-heaps.

3.1 Determining the Size of Sub-Heaps

Whenever our allocator needs more heap, it requests from the global memory manager a heap area whose size is a power of two. Our allocator associates the obtained heap area to a sub-heap. When that sub-heap is no longer required, our MemSave-VM releases the sub-heap back to the system. The size of a sub-heap strongly affects how much heap memory an application uses over the whole execution as well as at its peak. Since the MPU provides only four regions for an application and a newly obtained sub-heap is mapped for an MPU region, an application can have up to four sub-heaps. If sub-heaps are too small, applications may not secure the minimum heap size required for a successful execution. On the other hand, if they are too large, the memory usage will unnecessarily increase. In addition to covering the minimum heap size, we need to consider the lifetimes of sub-heaps. Since a larger sub-heap is less frequently freed than a smaller sub-heap, unduly large sub-heaps hamper the efficiency in heap sharing by reducing the chance of releases. We can make each sub-heap of an application have different size from one another, or alternatively, we can make all sub-heaps have the same size. In the former policy, we can follow the conventional principle that adaptively increases and decreases the allocated size by doubling and cutting in half. Using this strategy, we can raise the heap size to N only by $\log_2 N$ times of enlargement. This strategy, however, tends to allocate unnecessarily large heap at the peak of memory usage. In addition, since we can expand the heap size only by four times, we rather make all sub-heaps have the same size. We still make a distinction between sub-heaps for dynamic objects and ones for permanent objects by using different sub-heap sizes. This results in more efficient use of memory, since permanent objects are allocated in a different frequency.

We present an algorithm, called determine_subheap_size(), which determines the sizes of sub-heaps for dynamic objects and permanent objects. The algorithm is based on profiling information about the memory usage of an application such as the maximum size of live dynamic data, and the maximum size of live permanent data, and the size of the largest object. We can derive the amount of maximum live data, i.e. the peak live data over the execution, using profile runs where a garbage collection is invoked after every object allocation. The largest object size can be easily obtained by comparing allocation sizes at every allocation.

Determine_subheap_size() aims to pack the maximum live data into at most four sub-heaps since an application can have up to four regions. The largest dynamic object size provides a lower bound for a dynamic sub-heap size, since we need to allocate an object within a sub-heap. Determine_subheap_size() tries to use as many sub-heaps as possible, since many small sub-heaps can allow sharing in a finer granularity. The algorithm, however, checks if the small sub-heap can contain the largest object. If not, the algorithm increases the size of a sub-heap and check the constraint again. The solution generated by the algorithm provides the smallest size of sub-heap among all feasible configurations. The number of feasible configurations is at most three, since we only consider four, two and one sub-heap(s) by doubling the size of sub-heaps. Note that determine_subheap_size() always generates sub-heap sizes which are power of twos to map them to MPU regions.

In addition to the sizes of dynamic sub-heaps and permanent sub-heaps, determine_subheap_size() provides the size of *initial permanent area* which is a special area for permanent objects allocated in the first dynamic sub-heap. We basically segregate permanent objects from dynamic objects by putting them in different sub-heaps. Since the total size of dynamic sub-heaps should be a power of two, one sub-heap will have extra space even when the size of all dynamic objects reaches the maximum (*maxDynamic*). Thus, the algorithm tries to use that extra space for permanent objects. Since permanent objects are usually allocated during start-ups and stay alive until the end of executions, we will accommodate the extra space in the first dynamic sub-heap. Due to the initial permanent area within the first dynamic sub-heap, we often do not need any separate permanent sub-heap.

Algorithm 1 determine_subheap_size()

```
Input :  maxDynamic /* the maximum dynamic live data */
         maxPerm    /* the maximum permanent live data */
         maxObject  /* the size of the largest object */
Output:  dynheap_size  /* the size of dynamic sub-heap */
         permheap_size /* the size of permanent sub-heap */
         initpermsize  /*the size of initial permanent area */
 1: $dynheap\_size \Leftarrow$ ceil_power2($maxDynamic$)/4; /*ceil_pow
    -er2($X$) : the smallest power of two not smaller than $X$*/
 2: while (true) do
 3:     $current\_num\_subheaps \Leftarrow \left\lceil \frac{maxDynamic}{dynheap\_size} \right\rceil$ ;
 4:     $initpermsize \Leftarrow dynheap\_size \cdot current\_num\_subheaps$
                        $- maxDynamic$ ;
 5:     if $maxPerm > initpermsize$ then
 6:         increase $current\_num\_subheaps$ by 1 ;  /* another
              sub-heap is required for remaining permanent objects */
 7:         $permheap\_size \Leftarrow$ max(ceil_power2($maxPerm - init$-
                       $permsize$), $4 \times 1024$ ) ;
 8:     else
 9:         $permheap\_size \Leftarrow 4 \times 1024$ ;        /* we do not need a
                                              permanent sub-heap. */
10:     end if
11:     if $current\_num\_subheaps \leq 4$  and
              $dynheap\_size \geq maxObject$ then
12:         output $dynheap\_size, permheap\_size$ and $initpermsize$ ;
13:         break ;
14:     end if
15:     $dynheap\_size \Leftarrow dynheap\_size \times 2$ ;
16: end while
```

The details of determine_ subheap_size() is described in Algorithm 1. The following shows an example running of determine_subheap_ size() when *maxDynamic* is 110K, *maxPerm* is 40K, and *maxObject* is 10K.

ITERATION 1

- (Step i: line 1 of Algorithm 1) We find the smallest power of two not smaller than the given maximum dynamic live data, ceil_power2(*maxDynamic*) = ceil_power2(110K) = 128K. Thus, *dynheap_size* = 128K / 4 = 32K.

- (Step ii: line 3 and 4) We calculate the number of sub-heaps as

$$current\_num\_subheaps = \left\lceil \frac{maxDynamic}{dynheap\_size} \right\rceil = \left\lceil \frac{110K}{32K} \right\rceil = 4.$$

  We also set the size of initial permanent area as the difference between the maximum dynamic live data and the size of total dynamic sub-heaps.

$$\begin{aligned} initpermsize &= dynheap\_size \cdot current\_num\_subheaps \\ &\quad -maxDynamic \\ &= 32K \cdot 4 - 110k = 18K. \end{aligned}$$

- (Step iii: line 5 – 10) Since *maxPerm* = 40K > *initpermsize* = 18K, *current_num_subheaps* is increased to 5. ceil_power2(40K - 18K) = 32K. Thus, *permheap_size* is 32K.

- (Step iv: line 11 – 15) Since $current\_num\_subheaps = 5 > 4$, *dynheap_size* is doubled to 64K and next iteration begins by going back to Step ii.

ITERATION 2

- (Step ii)

$$current\_num\_subheaps = \left\lceil \frac{maxDynamic}{dynheap\_size} \right\rceil = \left\lceil \frac{110K}{64K} \right\rceil = 2.$$

$$\begin{aligned} initpermsize &= dynheap\_size \cdot current\_num\_subheaps \\ &\quad - maxDynamic \\ &= 64K \cdot 2 - 110k = 18K \end{aligned}$$

- (Step iii) Since *maxPerm* = 40K $>$ *initpermsize* = 18K, *current_ num_subheaps* is increased to 3. ceil_power2(40K - 18K) = 32K. Thus, *permheap_size* is 32K.
- (Step iv) Since $current\_num\_subheaps = 3 < 4$ and *maxObject* = 10K $<$ 64K, the resulting *dynheap_size* is 64K, *initpermsize* is 18K, and *permheap_size* = 32K.

3.2 Maintaining Free Blocks Across All Sub-Heaps

Our allocator uses a list of free blocks sorted in the order of starting addresses. Ordering the list by address is one of techniques already used in KVM. The slight difference of the free list in our MemSave-VM is that it contains all free blocks from all sub-heaps. When our allocator searches for an free block for allocation, it follows *first-fit* strategy which uses the first sufficiently large free block for allocation by searching the list from the beginning. An advantage of the address-ordered first-fit strategy is that the adjacency of two free blocks is automatically revealed by their addresses. This enables fast free block coalescing without additional space cost for boundary tags or double linking. Consequently, the address-ordered first-fit strategy decreases the minimum required size of an object [10]. Furthermore, the address-ordered first-fit strategy shows good cache behavior.

When the KVM allocator finds a free block larger than the requested size, it splits the block into two parts and uses the higher-addressed part for allocation. It then enlists the lower-addressed part back to the free-list. We modified our allocator to use the lower-addressed part for allocation and put the higher-addressed remainder back to the free-list. Consequently, the KVM allocator allocates objects from the highest address to the lowest address, while ours allocates from the lowest to the highest. Since compactors slide live objects from the highest address to the lowest address, allocating from the lowest address reduces the number of the copies of live data during compaction.

3.3 Releasing Surplus Sub-Heaps

Our garbage collector is a mark-sweep compact collector. Being aware that the heap area is composed of one or more sub-heaps, it releases some empty sub-heaps after GCs. Since our MemSave-VM maintains all sub-heaps in the order of their starting addresses, the mark phase is similar to that of KVM. Our sweep phase is also similar to that of conventional KVM except that two free blocks residing in separate sub-heaps should be carefully handled during coalescing. For compaction, KVM's mark-sweep compact collector applies a table based method which constructs a table of relocation information, called break table, in free areas. As the areas of live data are relocated towards one side of the heap (i.e. the lowest address of the heap), the break table should move in the opposite direction (i.e. towards the boundary between dynamic and permanent areas in Figure 5). While moving, the entries of break table are sorted if they are disordered. The break table algorithm preserves the allocation ordering of objects. Furthermore, it incurs no space overhead in storing relocation information within a contiguous heap area, if the size of smallest object in the heap is at least two-word which is the size of a break table entry [7]. There is always sufficient room to store relocation information in the free space within the heap. Since it is appropriate for a memory-constrained JVM, we also adopt the break table algorithm for compaction. Our compaction phase is, however, different in three ways:

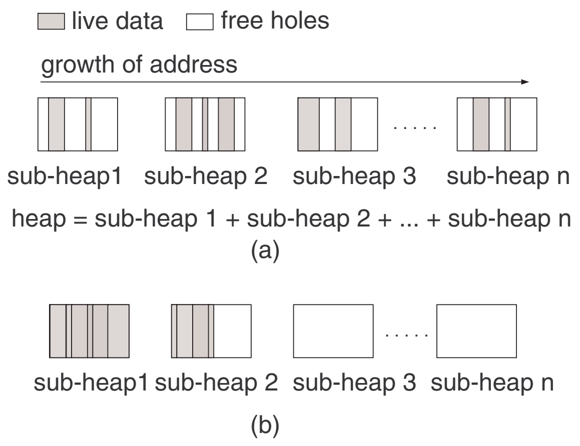

Figure 6: A compaction in our proposed method: (a) before compacting, (b) after compacting.

Compacting across non-contiguous regions To compact as much as possible, our compactor pushes all live objects towards the start of the lowest addressed sub-heap by moving live data of one sub-heap to another sub-heap (refer to Figure 6). When we compact the live data across non-contiguous areas, moving a chunk of live data can involve more than one entry of relocation information since the chunk can be moved to multiple sub-heaps divided into several sub-chunks as shown in Figure 7 (a). Consequently, we often cannot store all entries of relocation information in the free space within sub-heaps (refer to Figure 7 (b)). In addition, sorting the entries of a break table which is distributed among multiple sub-heaps often requires an extra contiguous space for the indirection to the entries. To minimize the space overhead, we modified the conventional break-table method into a two-step algorithm: (1) a local compaction process within a sub-heap for all sub-heaps and (2) a global compaction across all sub-heaps. The local compaction uses the same break table method described previously and incurs no space overhead. The global compaction also uses a relocation table constructed outside the heap, but the size of that relocation table is bounded by a small constant. This global compaction incurs at most $\sum_{n=2}^{4} n$ table entries, where $n(\leq 4)$ is the number of sub-heaps. Each sub-heap has at most one chunk of live data after the local compaction. In the global compaction, the compacted live data in the k^{th} sub-heap $(2 \leq k \leq n)$, can move to at most $(k-1)$ preceding sub-heaps divided into at most k sub-chunks. The last chunk can move within the k^{th} sub-heap. Thus, our algorithm needs at most nine additional entries outside the sub-heaps.

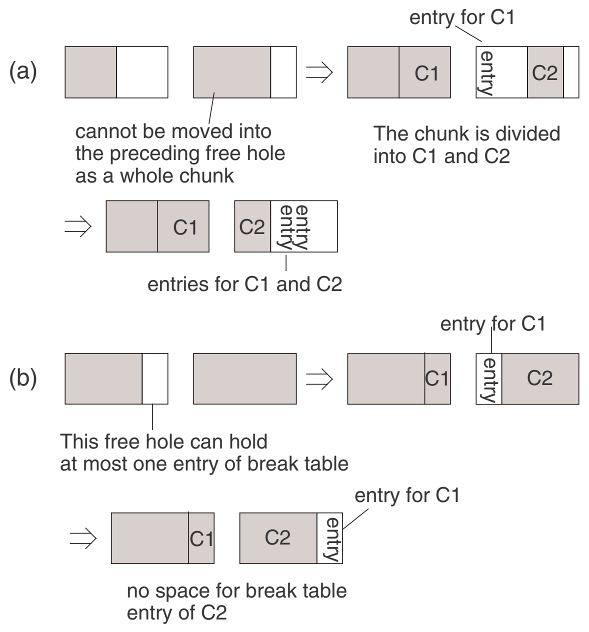
(a)
entry for C1
C1
entry
C2
cannot be moved into the preceding free hole as a whole chunk
The chunk is divided into C1 and C2
C1
C2
entry
entry
entries for C1 and C2
(b)
entry for C1
C1
entry
C2
This free hole can hold at most one entry of break table
entry for C1
C1
C2
entry
no space for break table entry of C2

Figure 7: Difficulties in compacting across fragmented heap using break table method: (a) when a live chunk is divided to move, (b) when the space for relocation information lacks.

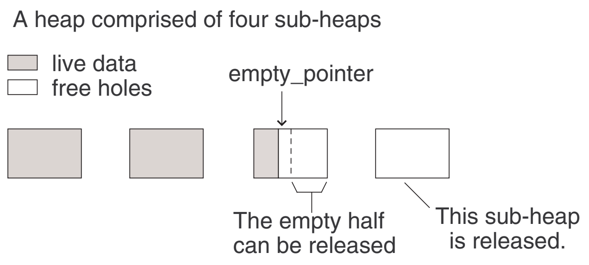
A heap comprised of four sub-heaps
live data
free holes
empty_pointer
The empty half can be released
This sub-heap is released.

Figure 8: Releasing empty spaces.

Releasing surplus space At the end of the compaction, our compactor knows which sub-heaps are empty. Our MemSave-VM releases all the sub-heaps that contain no live objects (refer to Figure 8). A sub-heap will contain live objects only in the partial portion of it. If the empty portion of the partially occupied sub-heap is greater than or equal to the half size of the sub-heap, we can have three release policies.

(1) Releasing the empty half after each GC

(2) Releasing the empty half when the sub-heap keeps it empty for a certain number of consecutive GCs

(3) Not releasing the empty half at all

Assuming the global memory manager follows the principles of the buddy heap system, we can readily free the empty half since the size of the half is also a power of two. While policies (1) and (2) more eagerly shrink the heap area than policy (3), they can incur more GCs or even request a new sub-heap too soon after the release. Policy (3) releases only the completely empty sub-heaps.

Triggering compactions The original KVM with mark-sweep compact garbage collector invokes a compaction only if it fails to find a free block for an allocation even after mark and sweep phases. The same compaction approach is inadequate for releasing the surplus heap areas since JVM does not voluntarily invoke compactions when it has enough heap space. Since finding surplus space large enough to release usually requires compactions, we investigate several alternative strategies for compaction.

(1) *Aggressive compaction*: compaction is invoked after every mark-sweep phase.

(2) *Periodic compaction*: compaction is invoked after every user-defined clock cycle.

(3) *Selective compaction*: compaction is invoked when compaction is necessary to allocate a new object, or when compaction can gather empty room large enough to release according to its estimation. The estimation is based on the total free blocks after mark-sweep phase. If the total free size is larger than or equal to the size of a sub-heap, we estimate that we have sufficiently large space to release.

We adopt the selective compaction strategy. The aggressive compaction is extremely inefficient. The amount of release in the aggressive compaction is similar to that in the selective compaction, while it consumes too much time in compactions. Setting a proper period for the periodic compaction is generally difficult since it will be an application specific parameter. In addition, the memory allocation behavior is usually uneven both in size and frequency. On the other hand, selective compaction strategy is effective in directing the releases of empty regions and also efficient in time.

4. EXPERIMENTAL EVALUATION

We implement our proposed dynamic allocator and garbage collector in the KVM within CLDC version 1.0.4 from Sun Microsystems [8]. We run our experiment on a Redhat 9 Linux PC equipped with a 1.8 GHz Pentium4. KVM uses a mark-sweep compact algorithm for garbage collection. Our garbage collector extends the existing mark-sweep compact algorithm to accomodate fragmented heap areas. We also extends KVM and MIDP 2.0 to accept several parameters we use in our experiment.

Through our experiment we measure the memory requirement of individual applications when they use our modified KVM to allocate heap segments in a shared pool. Since we individually measure each application, the functionality of the global memory manager is mimicked by malloc() and free() in C standard library. For the simulation of the MPU, we keep the region settings inside the global memory allocator. These settings will be passed on to the operating system and stored in the related registers to change the access criterion.

We conducted our experiments with six MIDP benchmarks. For interactive programs, we record and play back mouse and keyboard inputs using the MacroExpress from Insight software solutions [12].

4.1 Benchmark Characteristic

Table 1 shows a brief description of each benchmark. Table 2 shows memory related characteristics of each benchmark and the resulting configurations from determine_subheap_size(). The second column of Table 2 gives the maximum size of live objects. The numbers in parentheses distinguish the portions from dynamic objects and permanent objects. The third column shows the size of the largest object in each benchmark. These values are the parameters of determine_subheap_size(). Except for *hanoi* which uses around

Table 2: **The characteristic of benchmarks and the resulting configurations from** determine_subheap_size().

benchmarks	inputs to determine_subheap_size() maximum size of live objects (Dynamic/Permanent) (KB)	largest object size (KB)	outputs from determine_subheap_size() dynamic sub-heap size(KB)	initial permanent size(KB)	permanent sub-heap size (KB)
mreader	121.59 (93.98/27.61)	3.54	32	2	32
worm	46.27 (28.00/18.27)	3.54	16	4	16
bloodyghost	244.63 (163.64/80.98)	79.59	128	92	64
mdoom	214.96 (194.25/20.71)	32.02	64	61	4
hanoi	2036.24 (2027.97/8.27)	320.02	512	20	4
manyballs	62.57 (53.89/8.68)	3.14	16	10	4

Table 3: The comparison of memory usage between original KVM with mark-sweep compact collector and our MemSave-VM.

benchmarks	KVM with mark-sweep compact collector (KB * sec)	MemSave-VM with heap sharing (KB * sec)	# regions (dynamic/ permanent)	reduction in space × time (%)
mreader	42488	32650	4 (3/1)	27%
worm	2122	1538	3 (2/1)	27%
bloodyghost	60006	56342	2 (2/0)	6%
mdoom	65713	32939	3 (3/0)	50%
hanoi	75121	40099	4 (4/0)	47%
manyballs	16840	11521	4 (4/0)	38%
avg.			3.33	33%

Table 1: The brief description of benchmarks used in our experiments.

benchmarks	description
mreader	mobile browser [13]
wormgame	feeding-worm game [15]
bloodyghost	modern shooter, role-playing game [14]
mdoom	3D shoot-them-up, mini clone of the famous 3D Doom game [13]
hanoi	solving the tower of hanoi problem with 15 rings [15]
manyballs	watching balls moving on the screen [15]

2 GB of dynamic and permanent objects at maximum, most of the programs use about tens of KB or hundreds of KB at their peaks. Refer to the fourth column through the sixth column. The resulting sizes of sub-heaps from determine_subheap_size() are all power of twos, which allows us to map a sub-heap onto an MPU region. All the experimental results in this paper are obtained using these heap memory configurations in Table 2.

4.2 The Reduction of Memory Usage

Our goal is to reduce the total heap size used by a program during its overall execution. Since we need to take account of space and time, we integrate the heap size over the time spent in execution. Table 3 compares the memory usage between the original KVM and our MemSave-VM. The second and third columns present the integral of heap size for the original KVM and our MemSave-VM, respectively. The fourth column shows the total number of regions, and the numbers within parentheses present the number of dynamic regions and the number of permanent regions used during the whole execution on MemSave-VM. Our method successfully uses up to four regions. Note that four out of six benchmarks do not require any permanent sub-heap due to the initial permanent area within the first dynamic sub-heap. The last column shows the reductions in the integral of heap size over time. Our memory management scheme results in considerable reductions ranging from 6% to 50% (33% on average).

Figures 9 illustrates in detail how large heap the original KVM and the MemSave-VM use over the execution. A solid line denotes the heap size of our MemSave-VM, while a dashed line denotes that of the original KVM. A bar denotes the amount of live data before/after each garbage collection when MemSave-VM is used. Looking at solid lines, we see several step-ups and step-downs along the execution time line. These ups and downs represent acquisitions and releases of sub-heaps respectively. The areas below dashed lines and solid lines represent the amount of heap usage over the execution on KVM and MemSave-VM respectively. The difference in areas between dashed lines and solid lines shows the reduction in heap memory usage from MemSave-VM. In all benchmarks, MemSave-VM shows much smaller heap sizes than KVM. In addition, all of the benchmarks use no larger heap at their peaks on MemSave-VM than on the original KVM. Note that the peak memory usage of *worm* drops from 64 KB to 48 KB. This is more tight use of heap since the maximum size of live data for *worm* is 42.67 KB (refer to Table 2). The contiguous fixed-sized heap requires one 64 KB MPU region to accommodate 46.27 KB live data since the size of an MPU region should be a power of two. On the other hand, MemSave-VM utilizes three 16 KB regions to hold the same amount of live data. In a similar manner, the peak of *mdoom* also falls from 256 KB to 192 KB.

The changes in the heights of bars in Figure 9 reflect that the required amount of heap memory changes during the execution. These changes can be classified into several types. First, some changes occur from interactions with users. As users interactively adjust several features within applications, applications require additional heap memory or release surplus heap memory accordingly. *Mreader* and *manyballs* are representatives of the interactive applications. Depending on the length of article, *mreader* (mobile browser) requires different sizes of memory. Within the range of the number of balls *manyballs* offers, users can increase or decrease the number of balls in the middle of execution. Second, some applications use a lot of memory only during the start-ups and use much less memory after they are stabilized. Game programs such as *worm*, *bloodyghost* and *mdoom* show this pattern. Finally, some

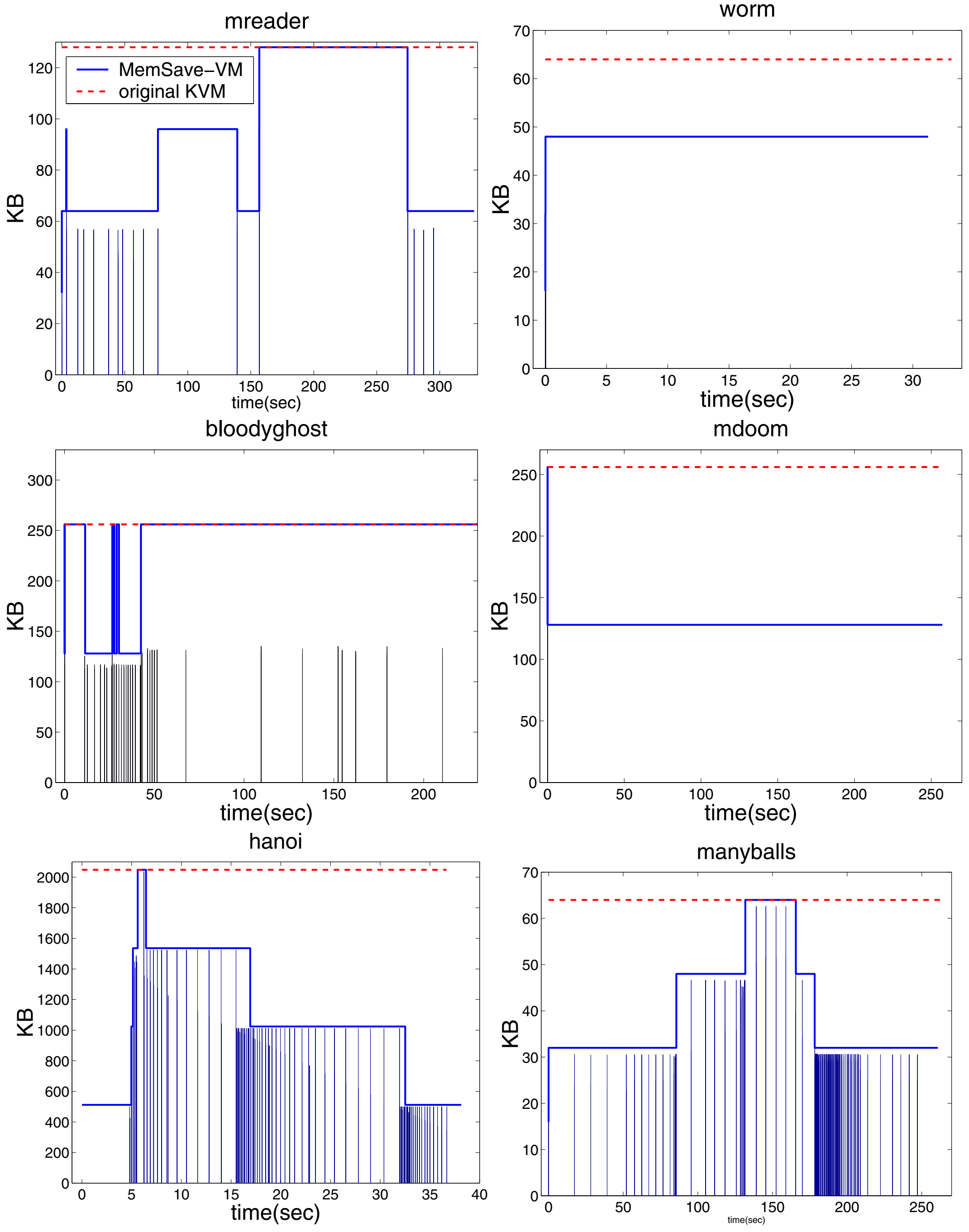

Figure 9: The usage of heap memory over the whole execution: a solid line denotes the heap size of our MemSave-VM ; and a dashed line denotes the heap size of original KVM; a bar denotes the amount of live data before/after each garbage collection when MemSave-VM is used.

Table 4: The total execution time and GC time.

benchmark	KVM with mark-sweep compact			MemSave-VM with heap sharing			perf.
	total (sec)	GC (sec)	GC frac.	total (sec)	GC (sec)	GC frac.	degradation
mreader	327.04	0.021	0.006%	327.25	0.066	0.020%	0.06%
worm	33.15	0.004	0.013%	32.06	0.003	0.008%	-3.29%
bloodyghost	234.40	0.051	0.022%	234.43	0.044	0.019%	0.01%
mdoom	256.69	0.006	0.002%	257.41	0.005	0.002%	0.28%
hanoi	36.68	0.149	0.407%	38.15	1.724	4.519%	4.01%
manyballs	260.50	0.011	0.004%	260.81	0.050	0.019%	0.12%
avg.							0.20%

Table 5: The number of garbage collections and the breakdown of the garbage collection time.

benchmark	KVM with mark-sweep compact					MemSave-VM with heap sharing				
	frequencies		time (sec)			frequencies		time (sec)		
	GCs	compact	mark	sweep	compact	GCs	compact	mark	sweep	compact
mreader	38	14	0.018	0.001	0.002	130	30(3)	0.054	0.003	0.009
worm	10	10	0.003	0.000	0.001	7	1(0)	0.002	0.000	0.000
bloodyghost	68	31	0.045	0.002	0.005	65	21(6)	0.037	0.002	0.004
mdoom	11	10	0.004	0.000	0.002	11	7(2)	0.003	0.000	0.002
hanoi	16	6	0.111	0.023	0.015	78	6(3)	1.367	0.173	0.184
manyballs	17	5	0.010	0.000	0.001	120	17(2)	0.044	0.001	0.005

applications are inherently dynamic in memory usage. *Hanoi*, to which a user gives nothing but one button click for initiation, shows this pattern.

Comparing the solid lines with the heights of bars, most benchmarks successfully adjust heap sizes according to the changes of the amount of live data. One exception is *bloodyghost*. Our heap management scheme is not quite effective for this benchmark. We cannot use smaller sub-heaps than 128 KB since the largest object size is about 80 KB. During the execution, we encounter the situation where half of the sup-heap is empty. Using an appropriate release policy discussed in Section 3.3 we should be able to release the empty half. For the convenience of our implementation, we choose not to release partially empty sub-heap. This is why we get the inefficient case for *bloodyghost*. In general, we keep heap space only the amount we need. Thus, the changes in heap usages enable us to share heap space among multiple tasks. For example, *mdoom* and *mreader* can simultaneously run within 256 KB heap memory, if *mreader* starts by obtaining the 128 KB heap after *mdoom* releases 128 KB surplus heap memory. Using a simple fixed size memory assignment, they require at least 384 KB heap memory to run simultaneously. Overall, the graphs reveal the effectiveness of MemSave-VM on reducing the memory footprint, by smartly utilizing the phased behavior of applications.

4.3 Impact on Execution Time

Our proposed method can incur runtime overhead since it needs to invoke garbage collector more often to be able to run on smaller heaps. Table 4 compares the total execution time and the GC time for KVM and MemSave-VM. The set of three columns under "KVM" respectively show the total execution time, the GC time, and the fraction of GC time over the total execution time. The set of three columns under "MemSave-VM" respectively show the counterparts of our MemSave-VM. The last column shows the performance degradation of our MemSave-VM compared to KVM. Table 5 shows the frequencies of GCs, the frequencies of compactions, and the breakdown of GC times. The numbers in parentheses of column eight refer to the frequencies of compaction phases that lead to the release of surplus regions.

The total execution times increase by 0.01% – 4.01%, while the execution time of *worm* decreases by 3.29%. The excution time of *worm* is improved since our scheme incurs fewer GCs and performed only one compaction while KVM does 10 compactions. The permanent objects in a separate sub-heap do not require compaction at all in our scheme. In contrast, KVM performs many compactions of dynamic objects to have more space for permanent objects during the start-up phases of applications. Furthermore, our MemSave-VM can spend a shorter time per allocation, since we have a smaller heap space to look up. Our scheme still can have many overheads that can nullify its good characteristics. For example, *hanoi* looses the performance. Since *hanoi* deals with a lot more dynamic objects than permanent objects (refer to Table 2), the numbers of garbage collections are greatly increased by the change of heap size. For remaining benchmark programs, the increases in the execution time are relatively small since only tiny portions of total execution times are spent in GCs (refer to the fourth and seventh columns in Table 4). Note that the frequencies and times of compactions using our scheme reveal our selective compaction strategy successfully adjusts the heap size incurring only a few time costs (refer to the eighth column of Table 4). To summarize our experimental results, our heap sharing scheme achieves considerable reductions in memory usage, while execution times increase only by 0.2% on average.

5. RELATED WORK

There are many research efforts that aim to reduce the memory usage to cope with the resource constrained Java environments. Griffin et al. [6] modified KVM to use reference-counting garbage collection scheme to foster faster reclamation of non-live memory blocks. They conducted experiments on several Java applications for embedded devices. Their work mainly aimed at the improvement in performance and energy consumption of an application program by reducing the total number of GCs. Chen et al. [5] proposed a GC-controlled leakage energy optimization technique of shutting off memory banks that hold no live data. They compared different compaction algorithms, different allocation schemes, and a couple of memory bank control schemes. They also estimated how those different compaction algorithms affected the energy consumption in system memory. Their experiments were done on KVM using several Java applications for embedded systems. Shaham et al. [1, 2] examined the potential improvement of execution time by freeing dynamic memory at the earliest possible points.

They developed a tool to rewrite application source code in a way that more timely garbage collection can be performed. They experimented with SPECjvm98 which is rather a representative benchmark for server applications. These existing research efforts help reduce the memory footprints of applications and the energy consumption in memory systems. They, however, do not dynamically adjust the heap size of an application at runtime in the context of multiprogramming. Yang et al. [4] proposed an adjusting method to find an optimal heap size that minimizes the number of GC invocations as well as page faults in the presence of virtual memory. Their method successfully tracks the memory footprint of a program during its run time and adjusts the heap size of the program according to the estimated footprint. They experimented with SPECjvm98 and pseudojbb benchmarks on the Jikes RVM. Their method mainly aimed at the improvement of execution time with the support of virtual memory from MMUs. Meanwhile, our scheme uses the memory protection units that have been actively used for guaranteeing protection of memory address space in multitasking embedded systems. Tual [11] proposed a generic architecture for multiapplication smart cards, which used an external MPU to access through domains to certain memory ranges.

Our memory management method is distinguished from these existing research efforts in that we focus on small memory-constrained but also multitasking embedded devices with no virtual memory. Our MemSave-VM reduces the memory footprints of an application by exploiting the phases in memory usage through sharing heap areas and deciding compaction timing carefully. Our heap sharing scheme also guarantees the protection of all fragmented memory areas by smartly utilizing MPUs.

6. CONCLUSIONS

We propose a heap memory management scheme for Java applications to reduce memory footprints in multitasking embedded systems with no virtual memory. Our MemSave-VM manages the fragmented heap areas acquired from a shared pool and also releases surplus heap areas back to the shared pool maintained by the global memory manager. Using the memory protection unit (MPU) and the help from the operating system, our scheme can protect all fragmented address spaces of an active task. Our experiments with six Java MIDP benchmarks show a significant reduction in memory footprints by 33% on average and experience only a slight increase in execution times by 0.2% on average. We believe reducing memory footprints is very important in small embedded systems. Our proposed technique can help increase the capabilities of embedded systems.

7. ACKNOWLEDGMENTS

This work was supported by grant No. R01-2006-000-11196-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

8. REFERENCES

[1] R. Shaham, E. K. Kolodner, and M. Sagiv. On the Effectiveness of GC in Java. In Proc. of *the 2nd International Symposium on Memory Management,* pp. 12-17, 2000.

[2] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap Profiling for Space-Efficient Java. In Proc. of *the ACM SIGPLAN 2001 conference on Programming language design and implementation,* pp. 104-113, 2001.

[3] A. Sloss, D. Symes, and C. Wright. *ARM System Developer's Guide.* Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[4] T. Yang, E. D. Berger, M. H. Hertz, S. F. Kaplan, and J. E B. Moss. Automatic Heap Sizing: Taking Real Memory Into Account. In Proc. of *the 4th International Symposium on Memory Management,* pp. 61-72, 2004.

[5] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection for reducing memory system energy in an embedded Java environment. *ACM Transactions on Embedded Computing Systems*, 1(1): 27-55, 2002.

[6] P. Griffin, W. Srisa-an, and J. M. Chang. An Energy Efficient Garbage Collector for Java Embedded Devices. In Proc. of *the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pp. 230-238, 2005.

[7] R. Johnes and R. Lins. *Garbage Collection: Algoriths for Automatic Dynamic Memory management*. John Wiley & Sons Ltd., West Sussex, England, 1996.

[8] Connected, Limited Device Configuration (CLDC) 1.0.4. [Online].
Available: http://java.sun.com/javame

[9] *KVM Porting Guide, CLDC, Version 1.0.4, Java 2 Platform, Micro Edition*, Santa Clara, CA, Sun Microsystems, Inc., 2002.

[10] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In Proc. of *International Workshop on Memory Management*, 1995.

[11] J-P. Tual. MASSC: a generic architecture for multiapplication smart cards. *IEEE Micro*, 19(5): 52-61, 1999.

[12] MacroExpress® [Online].
Available: http://www.macros.com

[13] midlet.org – Wireless Java Download. [Online].
Available: http://midlet.org

[14] Minoraxis, Inc. [Online].
Available: http://www.minoraxis.com

[15] Mobile Information Device Profile (MIDP) v2.0. [Online].
Available: http://java.sun.com/javame