Handles Revisited: Optimising Performance and Memory Costs in a Real-Time Collector

Tomas Kalibera Richard Jones
University of Kent, Canterbury
{t.kalibera, r.e.jones}@kent.ac.uk

Abstract

Compacting garbage collectors must update all references to objects they move. Updating is a lengthy operation but the updates must be transparent to the mutator. The consequence is that no space can be reclaimed until *all* references have been updated which, in a real-time collector, must be done incrementally. One solution is to replace direct references to objects with handles. Handles offer several advantages to a real-time collector. They eliminate the updating problem. They allow immediate reuse of the space used by evacuated objects. They incur no copy reserve overhead. However, the execution time overhead of handles has led to them being abandoned by most modern systems.

We re-examine this decision in the context of real-time garbage collection, for which several systems with handles have appeared recently. We provide the first thorough study of the overheads of handles, based on an optimised implementation of different handle designs within Ovm's Minuteman real-time collector. We find that with a good set of optimisations handles are not very expensive. We obtained zero overhead over the widely used Brooks-style compacting collector (1.6% and 3.1% on two other platforms) and 9% increase in memory usage. Our optimisations are particularly applicable to mark-compact collectors, but may also be useful to other collectors.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — Memory management (garbage collection); C.3 [*Special-Purpose and Application Based Systems*]: Real-time and embedded systems

General Terms Measurement, Performance, Algorithms

1. Introduction

A real-time garbage collector must address fragmentation. Some collectors do this by splitting objects (Jamaica [37], Sun's RTS [9]), some move objects (Aonix's PERC [30]) and some combine both techniques (Metronome [3], Fiji [34], Ovm [2]). A real-time collector must also be incremental so as to cause only short and bounded pauses of the mutator. This complicates object moving significantly. After an object is moved, all direct references to it must be updated but all mutator threads must see a consistent view of the heap at all times.

ISMM'11, June 4–5, 2011, San Jose, California, USA.

If objects are referenced directly — the case for most virtual machines today — reference updating is a lengthy operation that involves scanning the stacks, global data and all live objects in the heap. The mutator must to be allowed to run during this period. The use of direct references typically requires that both the old and the new copy of an object (or at least some portions of both) co-exist during that time, since the mutator may have references to both and the new location of an object is stored in the old copy. This copy reserve leads to a memory usage overhead similar to that of a copying collector [16] rather than that of a mark-compact collector [20]. A mostly non-copying collector [3] may copy fewer objects, but the worst-yet-unlikely-case overhead is still close to all live objects plus floating garbage.

If two copies of an object may co-exist, the system has to ensure that the application still runs as if there was only one. This transparency can be achieved with indirections. With Brooks forwarding [8], every read and write includes a dereference of a forwarding pointer stored in the object header. Every access is then performed on the *newest* copy of the object. With replication [19, 21, 29], any copies of a single object are kept in sync: writes are executed on *all* copies, while reads can proceed without any indirection. However, access to a field that is `volatile` or through an atomic primitive is problematic. Some systems remove the need for a copy reserve by calculating new addresses on the fly and moving objects on demand, but this requires operating system or hardware support and cannot provide real-time guarantees [11, 22].

In this paper, we explore how objects can be managed with *handles* [31], unmovable entities that represent objects and include a direct pointer to their contents. The heap, global data, and stacks use only handles to refer to an object, and thus all accesses are indirect. If an object is moved, only its handle needs to be updated, which is trivial and can be done atomically. Consequently, the old location of a moved object can be re-used immediately, without need for the copy reserve required by direct pointer implementations. Depending on how much of an object header is moved into the handle, some dereferences can be elided.

Handles have further advantages over direct pointers (not all of which are applicable to real-time systems). In contrast to direct references, no special action is necessary for pointer comparison. Because references are never updated, memory compression is easier [10], conservative collectors may move objects [31], and objects can be moved *at any time* (not just during a particular GC phase) or even multiple times during a cycle. The handle as a unique proxy to an object allows a hash code to be implemented simply as the handle address with no further space overhead [1] and allows objects to be represented even if swapped out to external storage or resident on a remote machine [18].

Handles have been used in the past for non real-time VMs, most notably by Sun's Classic and earlier VMs. Today most memory managers use direct references which are believed to offer better

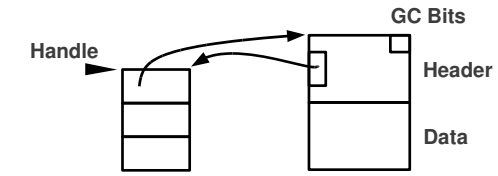
performance, and certainly do so for collectors that do not move objects while mutators are running. Real-time systems managed by incremental collectors change the trade-offs. These systems already incur the overhead of actions to support incremental updating of references *by the collector*: it is no longer self-evident that handles have to be more expensive than these actions. Handles offer the advantage that old copies of objects need not be kept. This is important not only because it can save memory but also because it makes worst-case memory usage easier to analyse, which is particularly nice for mostly non-copying collectors. On the other hand, the price for handles includes increased allocation time due to handle allocation and the memory overhead of reserved but currently not used handles. Implementation of the collector is also more complex. While some real-time collectors use handles [14, 27, 35], to the best of our knowledge, no publicly available study compares different handle designs or implementations, or compares these to direct pointers. We provide such a study here. In summary, our contributions are:

- Designs and implementations of different variants of handles in a (mostly non-moving) mark-sweep/compact collector in Ovm [32], an open-source real-time Java Virtual Machine.
- Optimisations to handle free-list manipulation and sweeping that significantly improve mutator and collector performance, and are applicable to any handle based mark-sweep/compact collector.
- An experimental comparison of different variants of handles with each other, and with Brooks forwarding and replication.
- We find that handles can keep up with other mechanisms for incremental, mostly non-copying collection. Our best configuration — of uni-sized fat handles — has no execution time average overhead compared to Brooks (1.6% and 3.1% on other platforms), 1.8% (5.2%) over replication, but needs 9% more collection cycles to run within the same sized heap.

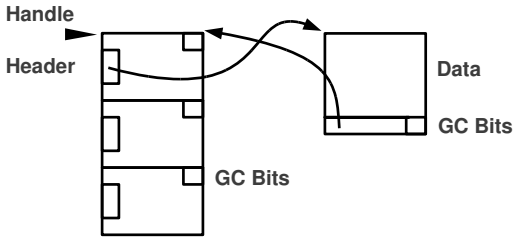
2. Design issues

Implemented naïvely, handles offer poor performance compared to other techniques for incremental compacting collection. However, there are many opportunities for optimisation. We consider the following.

- What information should handles hold? Should they include any header words as well as the pointer to the object?
- How should handles be allocated? The simplest solution would be to pre-allocate a fixed-size table but this makes the worst case utilisation of memory. Instead, we allocate and release *handle blocks* dynamically.
- Fragmentation is a significant issue for this handle space. Unlike objects, handles cannot be moved. Instead, we explore techniques that tend to avoid fragmentation. Can we encourage dense utilisation of handle blocks, thereby allowing other handle blocks to be recycled?
- There is substantial evidence [5, 15] that objects tend to live and die together. Can we take advantage of this both to reduce fragmentation in the handle blocks and to improve locality in the mutator?
- How can the collector trace and sweep both the handle and the data space in the most cache-friendly way?
- Some objects, including those in the boot image are known to be immobile. Can the compiler take advantage of this to short-circuit handle indirections?



(a) Thin handles



(b) Fat handles

Figure 1. Object layout of movable objects.

3. Handles

In this section we describe our handle design and optimisations in a way largely independent of Ovm. The design should thus be applicable to any mark-sweep/compact collector. Ovm-specific details are provided in Section 4 and performance implications are evaluated in Section 7.

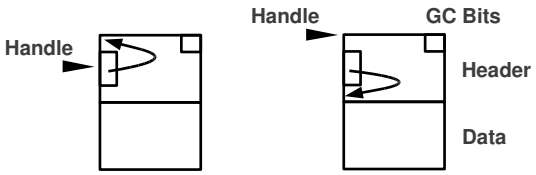
3.1 Object layout

We support three types of handles. *Thin* handles hold just a pointer to the object. Thus accesses to the object's header and fields require an indirection. By including the header in a *fat* handle (Figure 1), the indirection to access the header is removed but array and non-array handles now have different sizes, which makes it impossible to reuse handle slots between different object kinds. *Uni-sized fat* handles solve this by placing the array length with the payload rather than in the handle (header) — scalar and array handles have the same size but an indirection is needed for the array length field.

Compacting collectors sweep the object space (not the handle space) in order to discover objects to move. When an object is moved, its handle must be updated to refer to the new location. For this, we need to be able to find the handle from the object. Thus, objects hold a *back-pointer* to their handle.

Real-time systems cannot halt mutator threads to perform a collection, so objects are marked with a *colour* rather than a single bit [33]. All phases of collection require access to the GC colour. The colour is always part of the object header in Ovm, merged with the type information word. With thin handles, the header is already part of the object, so colours can be accessed during sweeping and compaction directly. Fat handles store the header (and hence the colour) in the handle. To avoid unnecessary cache traffic particularly while sweeping, we cache the colour in the back-pointer (Figure 1(b)). The back-pointer is used only by the collector, so the overhead of masking out the colour is not a problem. On the other hand, we do not cache colours in thin handles, because the speed-up we might gain in marking and so on would be outweighed by the slowdown of masking by the mutator.

The compiler can exploit knowledge that an object can never be moved. The object layout of unmovable objects is shown in Figure 2. This layout must be compatible with that of movable objects, because code may access both movable and unmovable ob-



(a) Thin handles (b) Fat handles

Figure 2. Object layout of unmovable objects.

jects. Unmovable objects thus still need handles. By keeping the handles with the objects, we exploit the likelihood that loading the handle into the cache will also load part of the header, thus reducing cache misses. The compiler can also replace a handle dereference by pointer arithmetic in code that is known to access only unmovable objects; this can eliminate completely some dereferences. Furthermore, unmovable objects do not need a back-pointer to their handle, as they are treated specially by the collector.

A consequence of having both movable and unmovable objects of the same type in the system is that, with fat handles, the header of an unmovable object has to be padded so that the payload starts at an 8-byte aligned address. This is because the payload of a movable object must start at an 8-byte aligned address and all fields of the object are laid based on this assumption. With uni-sized fat handles this is not a problem, because the header size is 16 bytes.

3.2 Handle allocation

We store handles in dedicated handle blocks, which are treated specially by the collector. Our handle space is not contiguous. Free handles are organised into a single-linked free-list, threaded through individual handles, which may be in different handle blocks. The allocator always takes the first handle from the head of the free-list. The handle space grows on demand — if no free handle is available, a new handle block is acquired, and new handles are initialised and linked to form a new handle free-list.

For fat handles, we maintain two independent handle spaces, one for scalar objects (smaller handles) and one for array objects (larger handles). To avoid the additional overhead of, and risk of fragmentation from, two handle spaces, we also implement uni-sized fat handles; we expect the overhead of the dereference to access the array length field to be relatively low, as this access will commonly be followed by ones to the array data, so there should be no extra cache load in most cases.

3.3 Handle release

When the garbage collector discovers a handle of a dead object, it has to return it to the free-list and zero the header part. We implemented different strategies on where to insert the handle, as we found that different choices have significant performance implications. The naïve yet simplest solution is always to add free handles to the head of the free-list (*naïve release*). Unfortunately, this solution has very poor performance, which we believe is caused by the resulting poor locality of this list. Consecutive handles in the list might well be in different handle blocks and the free handles of a particular handle block can be far apart in the list.

Consecutive free handles. A relatively simple but highly beneficial improvement is to keep all the free handles from a single handle block together in the free-list, not necessarily in address order. This rule can be enforced without any additional overhead on the allocator. For every handle block, the collector remembers the position of the last free handle in that block (the one that is last in the free-list). This information is not updated by the allocator, but only by the collector when it sweeps. When the collector needs to insert a newly freed handle into the free-list, it first checks the last known free handle in the respective handle block. The block for a handle can be found easily using address arithmetic. If this handle is still free, the collector inserts the newly freed handle in the free-list after that one. Otherwise, the newly freed handle is the only free handle in the block, so it can be connected to the head of the free-list. Discovering whether a handle is free is also relatively cheap. We maintain a bitmap indicating whether a given block is a handle or object block: a handle is free if and only if it points into a handle block. Most importantly, this test is needed only for sweeping by the collector and not for allocation by the mutator.

Free-list rebuilding. We support further optimisations to increase the locality of handles. After sweeping, we rebuild the free-list so that the free handles in a block are sorted in address order. Because all free handles in a block are stored consecutively in the list, this sorting can be implemented incrementally. The rebuild is just a linear pass through the free-list. Each time a new handle block is detected in the free-list, its sequence of handles is temporarily disconnected to avoid interference with mutator allocation, then incrementally sorted and re-connected. Note that the reconnection must be robust against the situation that the allocator has already consumed the previous handle block.

Free handle block release. We support opportunistic release of free handle blocks. While this is not a worst-case optimisation, handle blocks can be often released in practice. Just as objects tend to live and die together in clumps [5, 15], so do their handles. This optimisation is vital to reduce handle space fragmentation caused by peak demands on handles: allocation of many small objects that die quickly, followed by a period with a lower rate of allocation. Note that handle space fragmentation is worse for a real-time collector than a non real-time one. A stop-the-world collector can ensure that the number of used handles never exceeds a threshold unless all the handles belong to live objects; otherwise it triggers a collection. There is no such option in a real-time collector. An alternative is to pre-allocate the maximum number of handles that could be needed before the system runs out of memory. This approach, however, leads to excessive wasting of memory.

Sorting handle blocks. Our collector can sort the handle free-list while it is being re-built so that blocks with fewer free handles are included first (and hence re-used sooner by allocator). The motivation for this optimisation is to make the handle space more compact and to increase opportunities for releasing blocks of free handles. Sorting is performed incrementally in the linear pass of the free-list already done for rebuilding. We use a variant of bucket sort to find the location in the free-list at which to insert a given block, maintaining an array of blocks that records the last block in the free-list of given occupancy. As this array may be sparse, we use a bitmap to speed-up access to it. We also support reverse sort order, where mostly-free handle blocks are held first. This order may improve the locality of handle allocation, by giving longer runs of handles allocated from the same block.

Pre-initialised caching. If handle blocks are released as soon as all the handles they contain are free, and the handle space is nicely compact, new blocks for handles are needed soon. The time spent initialising them (and previously clearing empty handle blocks) is then wasted. Thus it makes sense to retain some empty handle blocks, initialised with a free-list threaded through each one. It would be possible to vary this threshold dynamically, but we have not done so. Keeping these in the free-list would result in repeated re-building if they survive a collection cycle. To avoid this, we can cache a certain number of empty yet pre-initialised handle blocks. When the allocator exhausts the handle free-list, it first tries to use a handle block from the pre-initialised list and only if that list is empty does it acquire and initialise a fresh block from the

system. Although this optimisation adds a branch, it is only to the allocator's (rare) slow path. Pre-initialised empty handle blocks are cached while rebuilding and perhaps sorting the free-list.

Colour caching. With both fat and uni-sized fat handles, we added a copy of the object colour into the payload, so that it can be accessed without dereferencing the back-pointer. This optimisation is important for sweeping, which is done in our collector by traversing the object space (not the handle space). Thus the sweeper does not need to access the header of live objects. We reduce cache traffic by updating the cached colour only when an object's fields are scanned (and hence must be loaded into the cache) rather than when the object (handle) is marked.

Independent handle release. Handles of dead objects need to be reclaimed. With thin handles, the sweeper does this (by following the back-pointer) as it discovers dead objects. This behaviour is not cache-friendly either. The handle free-list consolidation described above then runs *after* the object sweep has finished. With fat and uni-sized fat handles, the handle release can be done *while* sweeping objects, thereby improving locality and linearity of access of the sweeper. The sweeper passes through all used blocks of the heap in address order, sweeping both handle and object blocks. This handle sweep supports *consecutive free handles*. We also implement *free handle- block rebuilding* and *pre-initialised caching*. *Sorting* could be implemented as well, but instead we keep the blocks in address order, which could additionally improve locality. It also somewhat simplifies the handle sweep algorithm.

4. Minuteman RTGC and Ovm

Ovm [2, 32], developed at Purdue University, is an open-source implementation of the Real-time Specification for Java (RTSJ) [7] and a real-time garbage collector (RTGC). It compiles Java bytecode ahead-of-time into C, performing compiler optimisations including whole program analysis, devirtualisation, and inlining. The generated C code is then compiled by gcc to apply additional optimisations, including more inlining. The primary target platform is 32-bit Linux/x86, but Ovm has also been ported to the RTEMS/LEON embedded platform, used by the European Space Agency and NASA, and to other platforms.

Ovm is written mostly in Java, including the compiler, garbage collector, and runtime. At build time, Ovm runs in a hosted Java VM. It loads the bytecode for the whole VM and the application, compiles and optimises both, generates the C code, and then produces a final binary via gcc. It also generates a so-called boot image, which includes VM runtime classes, class information, and static data. We use a version of Ovm with green threading: all Java threads are run on top of a single native thread. Preemption points are inserted by the compiler only at back-branches. This has been shown to be sufficient to provide latencies below 6 μs [2] in response to external events including timer interrupts. Green threading greatly simplifies the collector: because the VM has control of scheduling it is straightforward to ensure that actions are atomic.

Minuteman is a highly configurable RTGC within Ovm. For this paper, we added handles as options for realising dynamic defragmentation. This implementation affected not only the GC, but other parts of the VM as well. We focus here on a GC configuration with arraylets, a fully incremental collector (all phases are incremental, but small objects are copied atomically), and GC barriers written for predictability (the barriers are on at all times, not just when needed by the GC). We use time-based periodic scheduling as in Metronome GC [3].

4.1 Memory layout and access in the boot image

As noted above, the memory layout and allocation is different for the heap and for the boot image. In the boot image, objects and arrays are always allocated contiguously, at 8-byte aligned addresses. The contiguous representation for the boot image is designed to be accessible using the same code as the heap representation. Thus, the forwarding pointers for Brooks and replication are set to point to the object itself, handles are part of the object representation as shown in Figure 2, and arraylets are laid out one after another to correctly reflect contiguous representation of the array. The compiler sometimes knows statically that a particular piece of code only accesses objects in the boot image. Such accesses are then optimised (eliminating arraylet pointer dereferences, skipping forwarding with Brooks, omitting double-writes with replication and optimising out dereferences with handles). These optimisations are important for performance because the boot image also includes type information needed for non-devirtualised calls and type checks, as well as data structures used by the memory allocator.

4.2 The heap

The heap is of fixed size, divided into 2 KB blocks. Each block can be free or dedicated to either small objects, arraylets, a large object, or handles. The kind of a block is stored in bitmaps and can be looked up at any time. Free blocks are always zeroed, organised in a single-linked free-list, and available to be allocated for any of the kinds. Small object blocks are allocated to size classes as in, for example, Metronome [3].

In fresh small-object blocks, new objects are allocated using a bump pointer. Once any objects in a block die, the block is no longer eligible for bump pointer allocation. Instead, free objects are organised into segregated free lists. There is a free-list per each small-object block, and a free-list of non-full small-object blocks of a given class. The small-object allocator preferentially allocates from a segregated free-list, and only if this list is empty does it acquire a new block and set it for bump pointer allocation. Small-object blocks that become free are zeroed by the collector and returned to the low-level block allocator. In general, zeroing is always done during the sweep in Ovm, to prevent pauses during allocation. Note that such pauses might affect the highest priority threads due to allocation activity of low priority threads. An adversarial allocation/lifecycle pattern may lead to very many almost-free blocks per size class; consequently, there may be no free blocks in the heap, and allocations may fail in another size class. This situation is prevented by dynamic defragmentation, which can move objects from one block to another within a size class. Arrays are formed by a spine which contains references to external arraylets (each the size of a heap block, marked as an arraylet block). The last arraylet can be allocated within the spine (as an internal arraylet) if it is smaller than 2 KB. Thus, arrays smaller than 2 KB have smaller access overheads. The spine is allocated either as a normal small object, if it small enough, or as a large object.

5. Modifications to support handles

In this section we outline the changes to the heap and to the write barriers to support handles.

5.1 Handle blocks

Each handle block holds only handles of one kind. Thin handles are 4 bytes long. Fat handles for scalars are 16 bytes and for arrays 20 bytes. Uni-size fat handles are 16 bytes. In addition to regular handle blocks that hold used and/or free handles, we keep pre-initialised handle blocks of free handles, sorted in address-order, which can be quickly re-used by the allocator. Free handles are allocated from a free-list, as explained in Section 3. Large objects do not use handle blocks but have their handles inlined in the same way as unmovable objects in the boot image.

In Ovm/Minuteman, the object header includes the type information word (with GC bits), a lock word, a pointer for RTSJ scopes,

the forwarding pointer, and a hash word (except in handle configurations). Thus, without handles, scalars have 20-byte headers, and arrays 24-byte; the hash word is initialised when the object is allocated. With handles, the hash code is the handle address so the hash word is redundant and the headers are 4 bytes smaller. Thus, compared with Brooks and replication, *there is no per-object overhead for handles* although unused handles may incur memory overhead in the handle space.

5.2 Barriers and dereferences

The collector uses a Yuasa style [39] snapshot-at-the beginning (deletion) barrier, which marks the old target of the overwritten field. Stacks are scanned on-the-fly, one by one — scanning a single thread's stack is atomic (recall that Ovm uses green threads) but the mutator can run before the collector scans the next thread. This incrementality requires the collector to also use a Dijkstra style [13] incremental update (insertion) barrier, which marks the target of the newly written reference. The collector uses card marking to record pages in the boot image that it needs to scan for references into the heap. Whenever a reference is stored into an image object, the barrier marks the appropriate bit for the respective image page.

Brooks forwarding. Any read or write from/to the heap has to be preceded by a forwarding pointer dereference. However, immutable header fields, including the type information and the hash word, can be accessed directly. Array access does not need a dereference, because array length and the pointers to the external arraylets are immutable, and the pointer to the internal arraylet is updated by the GC to point to the up-to-date version in the new spine. Before any reference is written to memory, it is forwarded to prevent spreading of old references during collection. An indirection is also needed for reference comparison.

Replication. Reads from objects do not require a dereference. Writes store their value to both replicas of the object. Arrays are accessed in the same way as with Brooks forwarding. References are again updated before they are written to memory. Minuteman also supports incremental object copying with replication, during which time the mutator holds only references to the old replica – but we use only atomic copying in this work. A dereference is also needed for reference comparison.

Thin handles. Any access to an object needs a handle dereference, be it to data or header. The hash code is the address of the handle, and thus can be obtained without a dereference. No updating is necessary when writing a handle, as handles cannot become old. Array access works exactly as in Brooks forwarding and replication, except for the initial handle dereference. Reference comparison is cheaper: it is just comparison of the handles' addresses.

Fat handles require fewer dereferences than thin handles, as all header fields can be accessed without a handle dereference. Fast access to type information (type checks and virtual method calls) is particularly important. The array length field is in the handle, so requires no dereference to access it (e.g. for bounds checking).

Uni-size fat handles behave like fat handles but also require a dereference to access an array's length.

6. The collection cycle

The collector is implemented in a single thread, which wakes up when free memory (the number of free blocks) is running low, runs a GC cycle, and goes back to sleep. The thread is interruptible at almost any time: the longest atomic operation is a copy of a small object (up to 2 KB). The VM can be configured for other operations not to process more than a given number of bytes atomically. The GC cycle is as follows. In each phase, we distinguish the actions required for different configurations.

1. `waitUntilMemoryIsScarce` The GC thread sleeps. All references are black, and objects are allocated black.
2. `scanStacks` The meaning of black and white is inverted, making all objects white. The allocation colour is made black. Thread stacks are scanned and any directly reachable objects are marked grey (this also stores the reference into a list of reachable not-yet-scanned references).
   Brooks forwarding and replication. Marking always uses the up-to-date location (forwarding the reference if necessary; replication identifies the old copy with a special bit). Note that references on the stacks cannot be updated yet because the heap may still include old references.
   Thin handles. Marking dereferences a handle.
   Fat and uni-sized fat handles. Need no dereference as they include the colour.
3. `scanImage` The marked (dirty) pages of the boot image are scanned for objects, which are in turn scanned for references to heap objects which are marked grey.
   Brooks forwarding and replication. References are also fixed to point to up-to-date locations.
4. `markAndCleanHeap` For each reference in the list of grey (reachable not-yet-scanned) references, the target object is removed from the list and scanned for references, which implicitly marks it black.
   Brooks forwarding and replication. The defragmentation phase of the previous GC cycle (see below) duplicated live objects but did not forward references. This was deferred to the mark phase of this cycle. Consequently, there may be references in the heap and stacks to both the old and the new location of a single object. Marking updates stale references.
   Fat and uni-sized fat handles. Mark handles of live objects; this colour is cached in the object only when it is scanned for child pointers.
5. `cleanStacks`
   Brooks forwarding and replication. Stacks are scanned again, just to update references to point to the new locations of objects. Handles do not require this step.
6. `sweep` All white objects are unreachable, and will be reclaimed. After the sweep phase, all objects in the heap are black. All references point to up-to-date locations (there are no old copies).
   Brooks forwarding and replication. All references in reachable objects now point to up-to-date locations. The old locations of relocated objects are reclaimed as they are white (unreachable). The external arraylets of a garbage array are released only when the non-old copy is swept (as we must not release them twice). Replication must update the forwarding pointer of the new copy to point to itself when the old copy is reclaimed.
   Thin handles. Handles of small objects require additional work. Those added to free-lists are kept linked to the objects. Thus, when the allocator acquires a free slot in an object block, it obtains a handle and back-pointer 'for free'. This optimisation would not work with fat handles, because scalars and arrays use different size handles (although scalar and array objects may be interleaved in the same size class). On the other hand, when a small object block is completely unused, the block is recycled and the handles of any objects that it contained are added to the handle free-list.
   Fat handles. All handles of dead objects are zeroed and released for re-use. With *independent handle release*, the handles are released independently of their objects. To do this reliably, the small object sweeper has to be able to distinguish an array from a scalar (without accessing the header that may already have been zeroed or re-used). In the case that it is an array, the

sweeper needs to free the external arraylets of the array. With fat handles, we distinguish arrays from scalars by the type of the handle block to which the back-pointer points as there are no bits spare for this purpose in the back-pointer. Thus, *independent handle release* cannot return a free scalar- or array-handle block to the system in case it is reused for a different purpose before we have swept all garbage objects whose handle was stored in that block. Instead, all freed handle blocks are pre-initialised.
Uni-size fat handles. We solve this problem with uni-sized fat handles. As these are all 16-byte aligned, we can use a spare bit of the back-pointer as an array indicator. This bit is initialised at allocation time. We can thus reliably detect arrays at any time, and return free handle blocks to the system, once a threshold of pre-initialised handle blocks has been used.

7. `rebuildHandleFreelist(s)` The handle free-list is rebuilt as described in Section 3. With all optimisations enabled, this means that the list is sorted so that handle blocks with fewer free handles are first, and that the free handles of a single block are together and in address order. Some free handle blocks are cached as pre-initialised for further allocation, and others are zeroed and returned to the allocator. This step is only needed with handles, and only in certain configurations:
Thin handles. Whenever the handle free-list should be rebuilt or sorted or free blocks be returned or pre-initialised.
Fat and uni-sized handles. Under the same conditions, but not for independent release. With fat handles, this step uses two passes, one for scalar handles and one for array handles.
8. `defragment` If the amount of free memory is below a given threshold, defragmentation is started, relocating objects from less occupied to more occupied blocks.
Brooks forwarding and replication. The old locations of objects remain reserved until the next sweep. Only then can they be reused for allocation. Atomically with object relocation, forwarding references are updated so that the old location points to the new location and the new location to itself. Note that evacuation is not guaranteed to release a block, as free slots in the block may be reused by the mutator while the collector is attempting to evacuate it, that is until the next sweep.
Replication. The copies' forwarding addresses point to each other, and the old location is also marked by a bit shared with the type information in the header.
All handles. Relocating an object updates its handle to point to the new location. Fully evacuated blocks are zeroed and returned to the system.
Thin handles. If an object is relocated to a location that already has a cached handle (see point 6 above), this handle is relocated to the old location of the object. And in case of successful evacuation of a block, it is then freed.

7. Evaluation

The primary goal of our evaluation is to discover how much slower handles are compared to the faster non-handle configuration of replication and Brooks. The secondary goal is to compare execution time and memory requirements of different handle configurations.

For our experiments we use a subset of the DaCapo 2006-10-MR2 benchmarks [6] that run with Ovm. We use periodic scheduling of the garbage collector as in Metronome: the collector is scheduled periodically, always in fixed duration time slots (500 μs), targeting mutator utilisation of at least 70% with a time window of 10 ms [3]. The collector, however, does not start a new cycle unless the free heap blocks account for less than half the heap. We run eager compaction, so all size classes are fully compacted at every GC cycle, since we find its cost negligible. Our statistical summary differs slightly with each experiment, but we always follow a common set of rules. We repeat every invocation of a benchmark (that is, start the VM binary multiple times) to average out noise due to memory placement. We iterate every benchmark within an invocation, as supported in DaCapo, to average out random perturbations of the system. We discard the first one third of measured iterations within each execution to limit the influence of start-up noise (note this is not possible with the experiments where the measurements are done by the VM, not the benchmark). We estimate the error bounds using non-parametric methods, and we round the results so as not to show digits invalidated by those error bars. When summarising relative overheads we use the geometric mean. When summarising execution times from different invocations and iterations of the same benchmark, we use the arithmetic mean, as the summary of execution time has a physical meaning [26].

	Linux	Intel platform	Clock	Cache
A	2.6.35	x86/64 Core2 Duo	2.4 GHz	4M 16-way L2
B	2.6.35	x86/32 Pentium 4	3.2 GHz	1M 8-way L2
C	2.6.32	x86/64 Nehalem/Xeon	2.27 GHz	8M 16-way L3

Table 1. Platforms used for experiments, including last level cache size.

We ran our experiments on three different platforms listed in Table 1. Note that while two platforms have multiple cores, Ovm only uses one. It has always been built as a 32-bit x86 executable.

Execution time overheads of handles. For this experiment, we choose the best configuration for thin handles, fat handles, and uni-sized fat handles, and we compare against replication (Table 2). The overhead of uni-sized fat handles (the best handle configuration) is about 1.8% over replication, 0.1% over Brooks, the error is within 2.3/2.4 percentage points respectively, and hence the slowdowns are not statistically significant (confidence intervals include 1, which is zero overhead). Fat handles seem slower than uni-sized fat handles, but the difference is not statistically significant (based on the confidence intervals in the table). However, we can conclude that thin handles are slower than other handles and direct pointers.

Memory usage with handles. Table 4 shows the total volume of objects moved during a benchmark execution (multiple iterations), measured in megabytes. The numbers are averaged over five executions (no error bars are shown as the numbers are extremely stable). The table shows that handles drastically reduce the number of objects moved. This is most likely because, with direct pointers, free slots in evacuated pages are often reused by the allocator, and hence after the next sweep the page is again only partially populated, and so once more is a candidate for evacuation. On average (not shown in the table), the amount of copied memory compared to replication is reduced to 4% (that is by 96%) with thin handles, to 5% with uni-sized fat handles and to 6% with fat handles. Fat handles move more data than other handles, probably because they use more GC cycles.

All the experiments for each benchmark were run with the same heap size. Handles are allocated in the heap. As the collector runs every time as soon as the heap is half-full, the number of GC cycles is a measure of memory requirements. Table 5 shows the number of cycles for the same experiment as in Tables 2 and 4. Numbers are again averaged over five executions, but left without error bars as they are extremely stable. Handles use more memory than direct pointers, but with thin handles the difference is tiny: they increase the number of GC cycles by only 1%. With uni-sized fat handles the number increases by 9% on average, which can be explained by the fact that they are larger, and thus the handle space fragmentation takes up a greater proportion of the heap. Fat handles increase the number of cycles by as much as one third — this is the fastest fat

	Antlr	Bloat	Fop	Hsqldb	Lusearch	Pmd	Xalan	**Geo-Mean**
Brooks	1.0328±0.0075	1.02 ±0.077	1.033±0.024	1.086±0.068	1.007±0.023	0.96 ±0.13	0.995±0.025	**1.018**±0.023
Uni-size Fat H.	1.0587±0.0072	0.973±0.084	1.021±0.019	0.999±0.056	1.046±0.061	1.0 ±0.1	1.06 ±0.03	**1.018**±0.023
Fat Handles	1.0815±0.0074	0.983±0.075	1.05 ±0.018	1.292±0.084	1.12 ±0.089	0.882±0.074	1.058±0.026	**1.061**±0.024
Thin Handles	1.0942±0.0071	1.25 ±0.15	1.053±0.026	1.152±0.074	1.214±0.087	1.11 ±0.13	1.081±0.032	**1.133**±0.032

Table 2. Execution time overheads of handles and Brooks forwarding. The baseline is replication [21]. Run on platform A.

	Antlr	Bloat	Fop	Hsqldb	Lusearch	Pmd	Xalan	**Geo-Mean**
Brooks	1.0393±0.0056	1.046±0.074	1.0217±0.0028	1.0014±0.0038	1.019 ±0.005	0.9955±0.0027	1.0212±0.0043	**1.021**±0.011
Uni-size Fat H.	1.0705±0.0054	1.05 ±0.08	1.0862±0.0025	1.013 ±0.002	1.0295±0.0054	1.0776±0.0027	1.0413±0.0073	**1.052**±0.012
Fat Handles	1.097 ±0.0053	1.026±0.083	1.076 ±0.002	1.173 ±0.013	1.0599±0.0056	1.1118±0.0051	1.0676±0.0049	**1.086**±0.013
Thin Handles	1.0974±0.0056	1.066±0.088	1.1051±0.0032	1.0768±0.0037	1.0982±0.0057	1.1612±0.0061	1.112 ±0.005	**1.102**±0.013

Table 3. Execution time overheads of handles and Brooks forwarding. The baseline is replication. Run on platform C.

handle implementation, which never returns free handle blocks to the system. Worse, fat handles need two handle spaces, thus adding to the handle space fragmentation.

It follows that the reduction in copy reserve for defragmentation does not outweigh the amount of memory wasted in unused handle slots. This is because although the amount of copying is reduced drastically, very few objects are copied anyway: in all configurations, there were at least 10,000× more objects allocated than moved. On the other hand, the amount of memory wasted in unused handles slots is very large (handle space utilisation is low).

The utilisation is the ratio of used handles to the current number of handles in the handle space. Unlike fragmentation in the object space which can be reduced by compaction, the handle space by its nature cannot be compacted. We aim to improve handle space utilisation by releasing free handle blocks. Out of the three selected configurations with best performance overhead (Table 2), thin and uni-fat handles release free handle blocks but fat handles do not (all blocks are pre-initialised). Further, fat handles use two handle spaces, which might worsen handle space fragmentation. Utilisation reported is the median of five runs, measured after every sweep; we report the geometric mean across benchmarks run on platform C. As expected, the utilisation with fat handles is worst (about 28% on average). Thin handles and uni-sized fat handles are better: 42% and 45% respectively. This is interesting: there are 4× as many handles per block with thin handles as with uni-sized, so one would expect far fewer opportunities to release a block with thin handles. But this is not the case: objects are probably allocated and are dying in long enough chunks [15]. Since the utilisation is nearly the same, the amount of memory wasted is almost 4× smaller with thin handles. This confirms Table 5 in that on average thin handles have much smaller memory overhead.

Execution time overheads on a platform with a larger cache. Table 3 shows relative execution time overheads measured on platform C (compare with Table 2 which ran on platform A). Brooks is faster than uni-sized handles, which are followed by fat and thin handles. The overhead of thin over fat is however within error bars. Compared to platform A, the overhead of fat and uni-sized handles seems larger (and is significant) while the overhead of thin handles seems smaller (and is not significantly larger than that of fat). We believe that this is because platform C has a very large last level cache (8 MB): with thin handles, more of the hot handles can remain in the cache. The average execution time overhead of uni-sized fat handles over Brooks is 3.1±1.2% (not shown in the table). On platform B, the overhead of uni-sized fat handles over Brooks is 1.6%±1.2 (significant, but seems smaller than on C).

Collector- and mutator-only overheads. Some of the overheads of handles over replication or Brooks shown in Table 2 can be attributed to the mutator (more complex allocation, more frequent dereferences, poorer locality in dereferences) and some to the collector (additional processing of the handle space, need for handles and/or back-pointer dereferences). The overheads exclusive to either the mutator or the collector are shown in Table 6. The results seem to suggest that the mutator overheads of handles may be higher than for Brooks, as expected, although the error bars overlap. However, the differences in collector overheads are large and significant, even given the errors. Best of the handles is the uni-sized handles configuration, which spends around 32% more time on GC than replication does (first column). This is partially because the GC runs more often, and partially because it is slower by about 24%: the second column normalises GC times by the number of cycles executed. The GC is faster with uni-sized fat handles than with thin handles, which is expected, because of *independent release* of handles and objects, which improves the locality of sweeping and also eliminates post-sweep handle space processing. It is a bit surprising that the GC with fat handles is also much slower than with uni-sized fat handles. This could be caused by poorer locality during collection due to the two handle spaces.

The average percentage of time spent in GC with the heap sizes we use is, however, small: 6% for Brooks and replication, 7% with thin handles and uni-sized fat handles, and 9% with fat handles. The GC time is dominated in all configurations by sweeping (50–60%, largely due to the cost of zeroing memory) and marking (30–40%). Stack scanning and updating each take less than 0.1% of GC time. Post-sweep handle processing (only thin handles in these experiments) takes about 5%. Compaction takes about only 0.4% (handles) to 1% (direct pointers) of GC time. Table 7 shows execution time overheads of marking, sweeping and compaction over replication. These numbers are normalised per cycle. The slowest sweep is with thin handles, which is not surprising because of their poor locality when releasing handles of dead objects. The sweep is about the same for fat and uni-sized fat handles. Handle marking is fastest with thin handles. This suggests that the overhead of the handle dereference to find the colour is smaller than the overhead of caching the colour when scanning objects of (uni-sized) fat handles plus the overhead of loading unneeded header fields of fat handles into the cache. Marking is much slower with fat handles than uni-sized fat handles. Compaction time per cycle is reduced to about 35% (that is by 65%) with thin handles, 46% with uni-sized fat handles and 56% with fat handles. Although (uni-sized) fat handles copy less memory per object because they do not copy headers, they copy more objects and hence more data (Table 4), and they take more time to compact.

Performance benefit of various handle optimisations. Execution time overheads of different configurations of uni-sized fat handles against the best one — *independent handle release* with *pre-initialised caching* (*Indep*) — are shown in Table 8. The fastest two configurations are *independent handle release* without *pre-initialised caching* (*IndepNopreinit*) as well as with caching all free handle blocks releasing none to the system (*IndepMaxpreinit*). The

	Antlr	Bloat	Fop	Hsqldb	Lusearch	Pmd	Xalan
Replicating	32.08	75.74	97.06	3.12	4.93	24.18	277.31
Brooks	33.65	74.96	99.87	3.15	4.93	24.4	270.81
Uni-size Fat Handles	2.69	0.84	0.66	0.06	2.17	0.97	18.27
Fat Handles	3.42	0.93	0.6	0.12	2.6	1.08	19.83
Thin Handles	2.04	0.81	0.57	0.04	2.16	0.78	15.86

Table 4. Total number of megabytes of data moved during defragmentation (platform A).

	Antlr	Bloat	Fop	Hsqldb	Lusearch	Pmd	Xalan	**Geo-Mean**
Brooks	1.0	0.97	1.01	1.0	1.0	1.0	1.0	**1.0**
Uni-size Fat Handles	1.19	0.94	1.06	1.07	1.04	1.29	1.09	**1.09**
Fat Handles	1.43	1.02	1.15	2.0	1.11	1.64	1.2	**1.33**
Thin Handles	1.02	0.93	0.97	1.0	1.01	1.1	1.02	**1.01**

Table 5. Number of GC cycles executed compared to replication (platform A).

	Collector	Col. per Cycle	Mutator
Brooks	1.045±0.015	1.05 ±0.015	1.021±0.029
Uni-size Fat H.	1.322±0.039	1.242±0.036	1.046±0.036
Fat Handles	1.78 ±0.03	1.393±0.026	1.049±0.031
Thin Handles	1.399±0.028	1.43 ±0.03	1.076±0.033

Table 6. Exclusive collector and mutator execution time overheads over replication. Relative mutator overheads are over the number of executed cycles. Run on platform A.

	Mark	Sweep	Compact
Brooks	1.14 ±0.016	0.999±0.015	1.065±0.037
Uni-size Fat H.	1.467±0.072	1.163±0.025	0.463±0.034
Fat Handles	1.8 ±0.034	1.18 ±0.02	0.555±0.023
Thin Handles	1.381±0.023	1.374±0.028	0.348±0.014

Table 7. Execution time overhead of GC phases, normalised per GC cycle. The baseline is replication. Run on platform A.

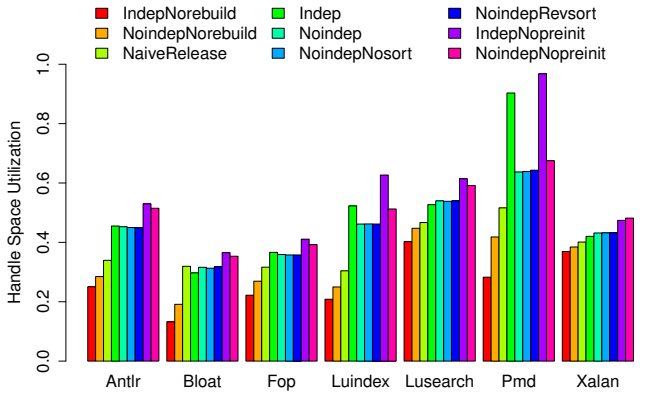


Figure 3. Handle space utilisation with different configurations of uni-size fat handles (platform B). Higher is better.

differences between *IndepNoPreinit*, *Indep*, and *IndepMaxpreinit* are negligible and within error bars, hence pre-initialisation does not make a difference on platform B. We have, however, observed a significant impact of the level of pre-initialisation on platform C and even more on platform A, where tuning the threshold (*Indep*) lead to the same overhead as with Brooks. *IndepNoRebuild* lacks *free-list rebuilding* and is slower than the previous configurations. The results show that *independent handle release* is also a good optimisation, as configurations lacking it are significantly slower (error bars do not overlap). Without *independent* handle release, *pre-initialised caching* does not improve the average overhead either (*NoindepNopreinit* is the same as *Noindep*, *NoindepMaxpreinit* seems even slower, but the difference is within error bars). By far and significantly the worst is *naïve releasing* free handles to the head of the free-list (*NaïveRelease*), which has overhead as much as 26% over the best version. This confirms that *consecutive free handles* are a crucial optimisation.

Sorting free handle blocks by occupancy has no performance effect: *Noindep* (free-list rebuilding, pre-initialised caching, sorting blocks by decreasing number of free handles), *NoindepNosort* (the same without sorting), and *NoindepRevsort* (the same, but sorting in reverse order) all have about the same average overhead over the best version and error bars overlap. *NoindepRevsort* has only slightly smaller average overhead than *NoindepNosort* and *Noindep*, the difference being well within the error bars. This suggests that sorting does not make a difference. *NoindepNorebuild* is a version with only *consecutive free handles*, but no *free-list rebuilding*. The average overhead seems higher than that of *Noindep* (sorting, pre-initialised caching), but it lies within the error bars.

Memory usage of various handles optimisations. We aim to increase handle space utilisation by releasing free handle blocks and by increasing the chance of free handle blocks (by sorting handle blocks). Median handle space utilisation from several executions with uni-sized fat handles is shown in Figure 3. The first three configurations (also first column of the legend) are those that do not release free handle blocks. The remaining six do: for every benchmark, releasing free handle blocks significantly improves the utilisation. Note that the actual values depend on allocation pattern of the application. The configuration giving highest utilisation is *IndepNopreinit*, which follows our expectations: it releases all free handle blocks. The configurations with maximum preinitialisation, *IndepMaxpreinit* and *NoindepMaxpreinit* (not shown in the graph) have very close utilisations to *NoindepNorebuild* and *IndepNorebuild*. It is not surprising that configurations that release free handle blocks also have better utilisation. With *preinitialised caching*, the utilisation is then very sensitive to the setting of the threshold of the maximum number of preinitialised blocks. For several benchmarks, configurations with *independent handle release* have better utilisation than without. This can be caused by the GC running faster, and thus fewer new handle blocks allocated during the GC. The graph also shows that *sorting handle blocks* does not make a difference: *Noindep*, *NoindepNosort*, and *NoindepRevsort* all have the same utilisation. Configurations with better utilisation run fewer GC cycles (not shown in the figure). This follows our expectation, as the GC is triggered when the amount of free memory falls below a given threshold. Increasing this threshold, and thus running the GC more often, should increase utilisation with all configurations.

	Antlr	Bloat	Fop	Luindex	Lusearch	Pmd	Xalan	**Geo-Mean**
IndepMaxpreinit	1.0135±0.0025	1.014±0.088	0.9851±0.0018	1.0098±0.0018	1.0093±0.0018	1.0352±0.0083	1.0025±0.0026	**1.009**±0.012
IndepNopreinit	1.003 ±0.002	1.031±0.086	0.9954±0.0026	1.0012±0.00095	1.0097±0.0025	1.0012±0.0052	0.9998±0.0033	**1.006**±0.012
IndepNorebuild	1.0179±0.0025	1.017±0.085	0.9901±0.0018	1.012 ±0.001	1.0133±0.0017	1.0478±0.0067	1.0012±0.0025	**1.014**±0.012
Noindep	1.0558±0.0023	1.05 ±0.091	1.0312±0.0025	1.0388±0.0013	1.0623±0.0054	1.165 ±0.011	1.0204±0.0029	**1.059**±0.013
NoindepMaxpre.	1.0636±0.0025	1.078±0.093	1.014 ±0.002	1.0444±0.0015	1.0602±0.0018	1.192 ±0.013	1.026 ±0.0026	**1.067**±0.013
NoindepNopre.	1.0549±0.0023	1.061±0.092	1.0138±0.0025	1.0415±0.0015	1.0536±0.0018	1.175 ±0.012	1.026 ±0.0039	**1.059**±0.013
NoindepNoreb.	1.0804±0.0026	1.048±0.098	1.036 ±0.002	1.056 ±0.002	1.077 ±0.0035	1.212 ±0.013	1.0263±0.0025	**1.075**±0.013
NoindepNosort	1.0534±0.0031	1.038±0.093	1.0303±0.0024	1.037 ±0.0015	1.054 ±0.002	1.16 ±0.01	1.0207±0.0029	**1.056**±0.013
NoindepRevsort	1.053 ±0.0026	1.016±0.093	1.0272±0.0025	1.0358±0.0012	1.0522±0.0016	1.161 ±0.012	1.0191±0.0026	**1.051**±0.013
NaïveRelease	1.1846±0.0032	1.25 ±0.11	1.1264±0.0021	1.23 ±0.002	1.2365±0.0073	1.848 ±0.068	1.0856±0.0031	**1.261**±0.016

Table 8. Execution time overhead of uni-size fat handles configurations against the best (Indep). Run on platform B.

8. Related Work

Recently, there has been renewed interest in handles for memory management, mainly for real-time and embedded systems [12, 14, 35, 38]. JOP [35] is a hardware implementation of a real-time Java VM for embedded systems with an incremental copying collector. The collector uses fat handles to support incrementality. Handles are allocated in a separate handle space of sufficient fixed size that it has enough handles for the worst-case demand of a particular application. Array and scalar handles have the same size — the array length field is used for the method table pointer in scalar handles. Each handle is either on a free-list or on a used-list. Allocation takes place from the free-list, and the allocator adds the new object to the used-list. Handle sweep operates on the used-list, adding unused handles to the head of the free-list. Handles allocated together will end up together on the list, if they also die together. It seems that no additional reorganisation of the handle free-list is performed. The handle dereference is implemented by hardware and can run in parallel with a corresponding null check and/or scope check. Fat handles with all metadata in the handles are argued to be good for memory mapped hardware structures, particularly arrays [36]. The payload does not include a back-pointer to the handle or any mark bits because collection is controlled from the handle space.

An alternative mark-compact real-time collector for JOP has been provided in [14]. It uses fat handles, allocated from a dedicated contiguous space that can grow but not shrink. The payload includes a back-pointer to the handle as well as to a mark-list. Free handles are organised in a free-list. Handles are allocated from its head, and added to the tail by the sliding compactor. Thus, although the handles freed in one collection cycle are appended in address order, overall the free-list is not address ordered.

Thin handles are used as a substitute for a mark stack in another mark-compact collector [38]. Outside a GC cycle, all pointers are direct. The GC replaces them by handles while scanning an object. An object is greyed by allocating a handle for it and linking it by a back-pointer. An object is blackened by removing that back-pointer. After sliding compaction, the handles are again replaced by direct pointers. The locality of the allocated handles should be good as it copies the live heap structure. The temporary nature of handles, on the other hand, leads to the need for not one but two rounds of pointer updating during a collection. Unlike most handle solutions, the addresses of handles cannot be used for hashing.

Compact-fit [12] is an allocator for explicit memory management which uses handles with size-class allocation and compaction. Compaction happens potentially after each object deallocation, with atomic update of the handle. Handles are allocated in a dedicated contiguous space and objects have back-pointers to handles. Free handles are stored in a free-list. Our optimisations for increasing handles locality should be applicable to compact-fit, together with the support for non-contiguous extensible handle space.

Handles are also apparently used in a garbage collector for BlackBerry devices [23]. The collector uses two memory spaces, RAM and flash. Handles were used in compressing mark-sweep and sliding mark-compact collectors [10] for Sun's KVM in order to eliminate the need for updating direct pointers in objects, which would require costly decompression of these objects. The handles were 8 bytes wide (pointer plus type information) and objects had back-pointers to handles. Arrays used arraylets and had array length in the spine. In the mark-compact version, the handle space was contiguous, expandable, but not shrinkable. One configuration eliminated handles by refraining from compression of reference fields. It turned out that this pays off: more memory is saved by avoiding handles than by compressing reference fields.

A hardware-assisted real-time collector with handles has been proposed in [27], using reference counting. Handles are allocated in a dedicated handle table and contain a reference count, the direct pointer and object size. It is not clear whether the design has been implemented or only simulated.

Another collector with handles has been implemented for Scheme [25]. The mark-compact GC uses thin handles, allocated out of the heap in dedicated chunks. Whenever a new block for objects on the heap is allocated, the GC also allocates a new handle chunk large enough to contain handles for all (the minimum size) objects that might be allocated in a heap block. Free handles are linked into a free list, but it is not specified how the list is managed. Movable objects have back-pointers to their handles. Unmovable objects have handles attached to their headers, as in our collector.

A two-space copying collector [31] used thin handles, allocated in dedicated handle blocks of 32 handles, outside the semi-spaces. Handles are used to allow conservative scanning with dynamic defragmentation. Free handle blocks are linked into a free-list and used (non-free) handle blocks are linked into a used-list. The handle space is swept by scanning the used-list, avoiding the free handle blocks. This is similar to the *pre-initialised caching* that we support. The total number of handle blocks in the system is fixed. As with [35], objects do not need back-pointers.

Handles have also been used in earlier Smalltalk implementations [24] and in early versions of Sun's JVMs [17, 28] in stop-the-world non-incremental collectors. Finally, the space overhead bounds for a system with segregated free-lists and partial compaction is discussed in [4], and the worst-case bound for a fixed-size handle space in [35].

9. Conclusion

Recently, there has been a renewed interest in handles for memory management and particularly for real-time garbage collection. Anecdotally, handles are believed to be very slow compared to direct pointers. We provide the first empirical comparison of handles to direct pointers (such as Brooks forwarding pointers) and describe optimisations to reduce their overhead. Our findings are that handles can be surprisingly fast with proper optimisations. We even obtained the same average performance as with Brooks forwarding (then 1.6% and 3.1% overheads on other platforms).

Our implementation is within Ovm's Minuteman RTGC. The results are applicable to mark-sweep or mark-compact collectors

with handles and some even to non real-time systems which use handles for distributed computation or persistence. A key optimisation to achieve such low overhead was a careful placement of free handles into a free-list: handles close to each other in memory should also be close in the free-list. This simple optimisation can be implemented without allocator overhead. In contrast, if free handles are added naïvely to the head of the free-list, the consequent overhead is very high — we measured 26%.

An important optimisation that reduced handle space fragmentation was releasing the memory occupied by unused handles. Still, the fragmentation can be around 40-60%. The amount of space consequently wasted depends on the handle size. When the handle was just a single pointer, we were able to run with almost the same number of collections as Brooks, but at a cost of 13% performance overhead (10% on another platform).

We show how to reduce the overhead of sweep by sweeping the handle space and data space independently in order to provide a more linear and local access pattern. The compaction phase of our collector (which is almost a reproduction of that in Metronome) is easier with handles than with Brooks: there is no copy reserve and many fewer objects need to be copied, resulting in much shorter compaction time. The lack of the copy reserve also means that it no longer needs to be part of the worst-case memory usage estimate, although the handle space size has to be included instead. Far shorter compaction time, however, does not make a difference overall in Ovm, because compaction is extremely rare in practice.

Finally, we thank the anonymous reviewers for their thoughtful comments and suggestions. We are grateful for the support of the EPSRC through grant EP/H026975/1.

References

[1] O. Agesen. Space and time-efficient hashing of garbage-collected objects. *Theor. Pract. Object Syst.*, 5:119–124, 1999.

[2] A. Armbruster, J. Baker, et al. A real-time Java virtual machine with applications in avionics. *Trans. Embedded Comput. Sys.*, 7(1), 2007.

[3] D. F. Bacon, P. Cheng, and V. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Principles of Programming Languages*, 2003.

[4] A. Bendersky and E. Petrank. Space overhead bounds for dynamic memory management with partial compaction. In *Principles of Programming Languages*, 2011.

[5] S. Blackburn and K. McKinley. Immix garbage collection: Mutator locality, fast collection, and space efficiency. In *Programming Language Design and Implementation*, 2008.

[6] S. Blackburn, R. Garner, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[7] G. Bollella, T. Canham, et al. Programming with non-heap memory in the real-time specification for Java. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[8] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Lisp and Functional Programming*, 1984.

[9] E. J. Bruno and G. Bollella. *Real-Time Java Programming with Java RTS*. Prentice Hall, 2009.

[10] G. Chen, M. Kandemir, et al. Heap compression for memory-constrained Java environments. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[11] C. Click, G. Tene, and M. Wolf. The Pauseless GC algorithm. In *Virtual Execution Environments*, 2005.

[12] S. S. Craciunas, C. M. Kirsch, et al. A compacting real-time memory management system. In *USENIX*, 2008.

[13] E. W. Dijkstra, L. Lamport, et al. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11). 1978.

[14] F. Gruian and Z. Salcic. Designing a concurrent hardware garbage collector for small embedded systems. In *Asia-Pacific Computer Systems Architecture Conference*, 2005.

[15] B. Hayes. Using key object opportunism to collect old objects. In *Object-Oriented Programming, Systems, Languages, and Applications*, 1991.

[16] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund University, 1998.

[17] C.-H. A. Hsieh, M. T. Conte, et al. Optimizing NET compilers for improved Java performance. *Computer*, 30, 1997.

[18] Y. C. Hu, W. Yu, et al. Run-time support for distributed sharing in safe languages. *Trans. Comput. Syst.*, 21, 2003.

[19] R. L. Hudson and J. E. B. Moss. Sapphire: copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3-5), 2003.

[20] R. E. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[21] T. Kalibera. Replicating real-time garbage collector for Java. In *Java Technologies for Real-Time and Embedded Systems*, 2009.

[22] H. Kermany and E. Petrank. The Compressor: Concurrent, incremental and parallel compaction. In *Programming Language Design and Implementation*, 2006.

[23] M. Kirkup. Taking out the trash: Garbage collection, 2005. `http://us.blackberry.com/devjournals/resources/journals/jan_2005/garbage_collection.jsp`.

[24] G. Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[25] M. Larose and M. Feeley. A compacting incremental collector and its performance in a production quality compiler. In *International Symposium on Memory Management*, 1998

[26] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[27] C.-M. Lin and T.-F. Chen. Dynamic memory management for real-time embedded Java chips. *Real-Time Computing Systems and Applications*, 2000.

[28] S. Meloan. The Java HotSpot performance engine: An in-depth look. `http://developer.java.sun.com/developer/technicalArticles/Networking/HotSpot/`.

[29] S. Nettles and J. O'Toole. Real-time replication garbage collection. In *Programming Language Design and Implementation*, 1993.

[30] K. Nilsen. Differentiating features of the PERC virtual machine. `http://www.aonix.com/pdf/PERCWhitePaper_e.pdf`, 2009.

[31] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In *Functional Programming and Computer Architecture*, 1987.

[32] Ovm. The Ovm virtual machine. `http://www.ovmj.net`.

[33] P. P. Pirinen. Barrier techniques for incremental tracing. In *International Symposium on Memory Management*, 1998

[34] F. Pizlo, L. Ziarek, et al. Schism: fragmentation-tolerant real-time garbage collection. In *Programming Language Design and Implementation*, 2010.

[35] M. Schoeberl. Scheduling of hard real-time garbage collection. *Real-Time Systems*, 45(3), 2010.

[36] M. Schoeberl, C. Thalinger, et al. Hardware objects for Java. In *International Symposium on Object-Oriented Real-Time Distributed Computing*, 2008.

[37] F. Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *Java Technologies for Real-time and Embedded Systems*, 2007.

[38] S. Stanchina and M. Meyer. Mark-sweep or copying? a "best of both worlds" algorithm and a hardware-supported real-time implementation. In *International Symposium on Memory Management*, 2007.

[39] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Systems and Software*, 11(3), 1990.