CINCOM Technical Note

VisualWorks Memory Management

CREATION DATE: OCTOBER 2003
REV DATE: MARCH 27, 2012
PRODUCT NAME: VISUALWORKS
VERSION: 7.9

This document describes the memory management strategies used by the VisualWorks virtual machine (object engine, VM).

This information is helpful when performance tuning certain memory-intensive applications. It is especially relevant for deploying applications with large working sets (e.g., one gigabyte or larger).

The facilities and policies described here are subject to change from release to release, so use this information with caution.

This technical note explores the following topics:

Memory Layout

At start-up, VisualWorks asks the operating system to allocate a portion of the available address space to house objects, native code and other resources, and then begins executing. Subsequently, VisualWorks may grow or shrink its memory usage dynamically.

VisualWorks runs as an operating system process with access to the full 32-bit or 64-bit address space made available to it by the operating system. VisualWorks is capable of using this address space as allowed by the operating system. Keep in mind that you may need to use a 64-bit VM to gain access to a 64-bit address space on 64-bit operating systems. Also, some 32-bit operating systems such as Windows may restrict the memory the 32-bit VM can allocate to less than 4GB.

VisualWorks makes a number of demands upon the address space. For example, each of the following can consume a fair amount of memory in the address space:

- Code and static data that belong to the object engine
- Dynamic allocations made by the C run-time libraries (such as stdio buffers)
- Dynamic allocations made by the window-system libraries
- Static and dynamic allocations made by the object engine

This section discusses only the algorithms associated with the last item.

The object engine manages two types of memory space: (1) a set of fixed-size spaces associated with the object engine and (2) a set of variable-sized spaces that comprise the Smalltalk object memory.

Fixed-Sized Spaces

The object engine allocates the following fixed-size memory spaces at system start-up time:

- Compiled Code Cache
- Stack Space
- New Space
- Large Space
- Perm Space

Currently, these spaces cannot change size once allocated, because the VM uses their location with respect to other spaces to determine the space that holds a particular object. However, the preferred initial sizes for these spaces can be set in the virtual image (VI) by sending sizesAtStartup:. Also, the -m1 through -m7 switches can override these preferences on the command line. See Setting Object Engine Space Sizes for more information.

Variable-Sized Spaces

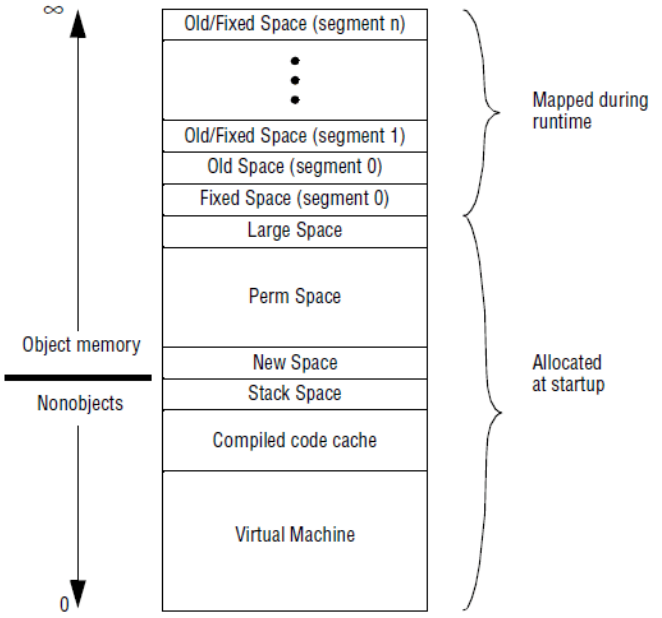
The object engine allocates an initial size for the following variable-sized spaces at start-up, sufficient to hold the existing old and fixed space objects, plus a free-space headroom in each.

- Old Space
- Fixed Space

As with the fixed spaces, the old and fixed space free space headroom can be controlled by sending a sizesAtStartup: message or using the -m1 through -m7 command-line switches (refer to Setting Object Engine Space Sizes). Note that old space headroom has to be large enough to allow VI initialization to the point that the VI memory policy is installed. If the headroom is too small, you may experience a VM out of memory failure during image startup.

Memory Organization

Each of these fixed- and variable-sized spaces is used by the object engine to house program elements of a particular type. The object engine organizes these spaces as shown below:



The diagram shows organization of the address space that belongs to a VisualWorks process, with code, non-objects, and the fixed portions of the object memory in the lower portion of the space, and the dynamic spaces and segments of the object memory above. Note that, when the address space is large enough, perm space may be mapped above all old and fixed segments instead.

Compiled Code Cache

To avoid the overhead of interpreting bytecodes, the object engine compiles each Smalltalk method into the platform's machine code before executing it. The compilation is automatic and transparent to the user.

When a Smalltalk method is invoked for the first time, the object engine compiles it and stores the resulting machine-code in the Compiled Code Cache, so that it can be executed. Once executed, this method's machine-code is left in the Compiled Code Cache for subsequent execution.

As its name suggests, this space is only used as a cache. If the cache runs out of space, those methods that have not been executed recently are flushed from the cache. This approach gives Smalltalk much of the speed that comes with executing compiled code, most of the space savings, and all of the portability that come with interpretation.

The size of this cache varies, depending on the density of the platform's instruction set. Default sizes are 640 KB for platforms with CISC-based processors and 1152 KB for 32-bit RISC platforms, and 1728 KB for 64-bit RISC platforms. These sizes are large enough to contain the machine-code working sets of most applications. The size can be changed at at image startup, as described under Setting Object Engine Space Sizes (below), to a maximum of 16 MB.

The compiled code cache is also used by the garbage collector to store its mark stack (for details, see Global Garbage Collector below), and to decompress compressed virtual image files on start-up.

Stack Space

Each process that is active in the virtual image (VI) is associated with a chain of contexts. Contexts are stored in two forms: the standard object format and the frame format.

When a Smalltalk program tries to send a message to or access an instance variable of a given context, that context must be in standard object form and housed in object memory. If it is not already in standard object form, then it is converted. The conversion to and from standard object format is transparent to the user.

On the other hand, when the method associated with a given context is actually being executed, that context must be in frame format and housed in the Stack Space. Once again, the conversion to and from this

form is automatic and transparent to the user. The frame format of the contexts has been designed to mate well with the typical machine's subroutine-call instructions.

The Stack Space is used as a frame format context cache. If there is not enough room in the Stack Space to store all of the contexts of all of the active processes, then the object engine converts some of these contexts to standard object form and places them in object memory to clear some Stack Space. When the system needs to execute the methods associated with these contexts, it converts the contexts back to frame format and places them back in the Stack Space.

The default size of this space varies from 20 KB to 40 KB, depending upon whether a given platform handles interrupts in another region of memory, or whether it needs to handle these interrupts in Stack Space. The size can be changed at image startup, as described under Setting Object Engine Space Sizes below. You can reduce the size of the Stack Space, at the cost of forcing the object engine to convert contexts more frequently from frame format to standard object format and back again. Or you can increase its size, at the cost of the additional memory.

Note that if the size allocated to Stack Space is significantly less than that required to hold the contexts for the active processes, applications may experience a stiff performance penalty. In some artificially demanding scenarios, code using a large number of Smalltalk processes may execute up to 10 times slower. The performance penalty is caused by two side effects of Stack Space being too small. First, context objects must be created so that Stack Space can be allocated to a process requiring execution. In addition, context objects for processes still requiring execution must be brought back into Stack Space. Second, after a process unwinds, any context objects associated with it must be garbage collected. To ensure your application is executing efficiently, you can use the Time Profiler (in the Advanced Tools parcels) or the MemoryMonitor contributed parcel, and examine the count of stack spills. While a few stack spills are acceptable, you should not see excessive thrashing of Stack Space. After increasing the size of Stack Space (see Setting Object Engine Space Sizes below), you should ensure that the change results in better performance. Increasing the size of Stack Space too much will have a small detrimental effect on performance.

New Space

New Space is used to house newly-created objects. It is composed of three partitions: an Object Creation Space, which we call Eden, and two Survivor Spaces.

When an object is first created, it is placed in Eden. When Eden starts to fill up (i.e., when the number of used bytes in Eden exceeds a threshold known as the scavenge threshold), the system's scavenging mechanism is invoked. Objects that are still reachable from the system roots are placed in whichever Survivor Space happens to be unoccupied at the time (one is always guaranteed to be unoccupied). Thereafter, objects that survive each scavenge are shuffled from the occupied Survivor Space to the unoccupied one. When the occupied Survivor Space itself begins to fill up (i.e., when the number of used bytes in the occupied Survivor Space exceeds a threshold known as the tenure threshold), the oldest objects in Survivor Space are moved to an object memory area called Old Space. When an object is moved from New Space to Old Space, it is said to be tenured. Both the scavenge threshold and the tenure threshold can be set dynamically (see Setting Object Engine Space Sizes for details).

The default size of Eden is 300 KB, and each Survivor Space (they are always identical in size) is 60 KB. These sizes can be changed at at image startup, as described under Setting Object Engine Space Sizes.

Large Space

Large Space is used to house bodies of large byte objects (bitmaps, strings, byte arrays, uninterpreted bytes, etc). The VM considers object bodies "large" when they are at least 1KB in size.

When a large byte object is created, its header is placed in Eden and its data in Large Space. This arrangement permits the scavenger to move the object's header from Eden to a Survivor Space without having to move the object's data. In fact, the data that is housed in Large Space is only moved when Large Space is compacted, as part of a compacting garbage collection or to make room for another large byte object, or in preparation for a snapshot.

If there are too many large object bodies to fit in Large Space, older ones are moved to object memory proper.

When the amount of data housed in the Large Space exceeds a threshold known as the LargeSpaceTenureThreshold, the scavenger is informed that it should start to tenure the headers of large objects. During the next scavenge, the headers of the oldest large objects are tenured to Old Space. However, the data of these large objects will not be moved from Large Space until the allocator actually runs out of space in Large Space. Only at that time will the data of these older large objects be moved to Old Space. The LargeSpaceTenureThreshold can be set dynamically.

The default size of Large Space is 200 KB. Again, the size can be changed at image startup, as described under Setting Object Engine Space Sizes.

Note that Large Space only houses object bodies, and that the headers for those bodies must reside in New Space, Old Space, or Perm Space. Thus, based on header location, large objects can be regarded as new, old, or perm objects.

Perm Space

Perm Space is used to hold all semi-permanent objects. Because they are rarely garbage, the objects housed in Perm Space are exempt from being collected by any of the reclamation facilities other than the global garbage collector. By removing such objects from Old Space, the time required to reclaim the garbage that may be present in Old Space is may be reduced according to the ratio of perm space size to old space size.

In the delivered virtual image, most of the objects in the system are housed in Perm Space. Newly created objects that are placed in Old Space by the scavenger are not automatically promoted to Perm Space.

Developers can move Old Space objects into Perm Space (and thus improving the efficiency of garbage reclamation) by creating an image by choosing **File > Perm Save As...** in the VisualWorks Launcher window.

For details, see Promoting Objects to Perm Space (below).

Smalltalk Object Memory

In addition to the above fixed-size memory spaces, the object engine also manages two variable-size spaces known as Old Space and Fixed Space. These spaces are warehouses for all objects that are not housed in one of the fixed-size spaces described above.

Old Space

Unlike the above spaces, the size of Old Space is not frozen at startup time. Instead, it is configured at startup time with a reasonable amount of free headroom space to install the VI memory policy. When Old Space begins to run short of free space, the system has the option of increasing its size. This growth is accomplished by means of a primitive that attempts to acquire additional address space from the operating system. The decisions regarding when to grow Old Space and by how much are controlled by a memory policy object. Until VisualWorks 7.7, the default memory policy was enforced by instances of MemoryPolicy,

which is now obsolete. In releases 7.7.1 and later, the default memory policy is enforced by instances of LargeGrainMemoryPolicy. We also provide the class MediumGrainMemoryPolicy. These memory policy classes are subclasses of AbstractMemoryPolicy. See the AbstractMemoryPolicy class comments and Memory Policies for details.

Although Old Space may be thought of as a single contiguous chunk of memory, it is implemented as a linked list of segments occupying the upper portion of the system's heap. Old Space's growth capability dictates this approach because, for example, I/O routines frequently allocate portions of the heap for their own use, creating intervening zones that divide Old Space into separate segments. In a growing system, then, Old Space may be composed of multiple segments. When these multiple segments are written out at snapshot time, they are stripped of their free space to conserve disk space. In addition, to eliminate fragmentation, the segments are coalesced into one large segment when the snapshotted image is loaded back into memory at startup time.

Each Old Space segment is composed of two parts: an object table (OT) that is used to house the old objects' headers, and a data heap that is used to house the objects' data. The data heap is housed at the bottom of the segment and grows upward; the object table is housed at the top of the segment and grows downward. Both the object table and the data heap are compacted by the compacting garbage collector.

Because the object table and the data heap grow toward each other (thereby consuming the same block of contiguous free space from different directions), the system should never run out of space for new object headers while still having plenty of space for object data, and vice versa. There is no arbitrary limit on the total size of Old Space, the total size of a given Old Space segment, or the number of Old Space segments that can be acquired. The only memory-related resource that the system can run out of is address space. On real-memory machines, this translates to available real memory. On virtual-memory machines, it corresponds to available swap space.

In addition, the system maintains a threaded list of free object table entries and a threaded free list of free data chunks. The incremental garbage collector recycles dead objects by placing their headers and bodies on these lists, and the Old Space allocator tries to allocate objects by utilizing the space on these lists before dipping into the free contiguous space between the object table and the data heap of each segment.

Note that Old Space must always have enough free space to account for at least a full New Space (eden space plus a survivor semi-space), a full Stack Space (the size required to create contexts for all frames in Stack Space), plus perhaps growing the remembered tables. This amount of memory represents the worst case scavenge, and it is reserved to ensure that the object engine can perform at least one scavenge in extreme low-space conditions, thereby providing the system with one final opportunity to take the appropriate action.

The same amount of memory should also be sufficient during image startup in the vast majority of circumstances. Therefore, at image startup, the default size of Old Space is set to the amount needed to house the old space objects in the image plus this amount of free space headroom. The headroom size can be changed via sizesAtStartup: or the -m1 through -m7 switches, as described under Setting Object Engine Space Sizes.

Fixed Space

Fixed Space is used to hold byte objects bodies whose data must not move. This is a requirement for data passed through the threaded API (THAPI), since threaded calls may be in process concurrently with a garbage collection. In general, fixed byte object bodies may be useful when used as communication buffers for libraries that assume the buffers do not move in memory. Fixed byte object bodies are not relocated during the object's life, but the space is reclaimed when the object is garbage collected.

New Fixed Space segments are added as needed, as with Old Space. Because the contents of Fixed Space cannot move, Fixed Space cannot be compacted, and so quickly becomes fragmented. When the image is saved, these multiple segments are stripped of their free space to save file space. They are then coalesced into one large fixed segment when the image is loaded back into memory at startup. Thus, because Fixed Space is coalesced at image start-up, it can be compacted by saving the image, quitting, and then restarting.

Object data ends up in fixed space if it is either:

- allocated explicitly, or
- passed as an argument of a threaded call.

The default size of fixed space is 200 KB. Again, the size can be changed at image startup, as described under Setting Object Engine Space Sizes (below).

Note that Fixed Space only houses object bodies, and that the headers for those bodies must reside in New Space, Old Space, or Perm Space. Thus, based on header location, fixed objects can be regarded as new, old, or perm objects.

Remembered Tables

Remembered Tables are structures used by the garbage collector to track references between various spaces. They are housed in Old Space.

The remembered table (RT) is a special table that contains one entry for each object in Old Space or Perm Space that is thought to contain a reference to an object housed in New Space.

The objects in the remembered table are used as roots by the scavenger. Thus, if an object is not transitively reachable from either the remembered table or the Stack Space, it will not survive a scavenge. The remembered table is expanded and shrunk as needed by the object engine. It is expanded if the object engine tries to store more entries than the RT can currently house, and it is shrunk during garbage collections when it has become both large and sparse, which can occur if a large number of entries were added and subsequently removed.

The old remembered table (OldRT) is a special table that contains one entry for each object in Perm Space that is thought to contain a reference to an object housed in Old Space. In addition, the OldRT keeps a list of Perm Space objects that have bodies stored in Old Space (perhaps as a result of become: or similar operations). This is so that the Old Space bodies of Perm Space objects are not collected.

The objects in the OldRT are used as roots by the incremental garbage collector and the compacting garbage collector — if an object is not transitively reachable from the OldRT, it will not survive a garbage collection. The OldRT is expanded and shrunk as needed by the OE. It is expanded if the OE tries to store more entries than the OldRT can currently house, and it is shrunk during garbage collections when it has become both large and sparse, which can occur if a large number of entries were added and subsequently removed.

The OE must be able to find enough contiguous memory in old space to grow the RTs. Thus, one of the responsibilities of the memory policy is to ensure there is always enough contiguous memory available for this purpose.

Facilities for Reclaiming Space

The object engine has several facilities for reclaiming the space occupied by objects that are no longer accessible from the system roots:

- Generational scavenger
- Incremental garbage collector
- Compacting garbage collector
- Global garbage collector
- Data compactor

Except for the scavenger, the object engine does not invoke these facilities directly. Policy decisions such as these are controlled by the VI. See the ObjectMemory and AbstractMemoryPolicy classes for the default policies, and the Memory Policies section.

Collector	Memory Spaces Reclaimed
Scavenger	New Space Large Space Fixed Space
Incremental GC	New Space Large Space Fixed Space Old Space
Compacting GC	New Space Large Space Fixed Space Old Space
Global GC	New Space Large Space Fixed Space Old Space Perm Space

Generational Scavenger

The primary reclamation system is a generational scavenger. The scavenger flushes objects that expire while residing in New Space. Typically, more than 95% of objects are reclaimed by the generational scavenger.

Briefly, the scavenger works as follows. Whenever Eden is about to fill up, the scavenger is invoked. It locates all of the objects in Eden and the occupied Survivor Space that are reachable from the system roots. It then copies these objects to the unoccupied Survivor Space. Once this copying is done, Eden and the formerly occupied Survivor Space contain only garbage, and so they are effectively empty and can be reused. The scavenger uses the objects in the remembered table and the objects referenced from the Stack Space as roots.

The scavenger's operation is imperceptible to the user under most circumstances. To ensure that this is so, the sizes of eden and the survivor semi-spaces should be limited so that the scavenger will start to tenure objects from New Space and place them in Old Space before the scavenger starts unnecessarily copying objects that will not become garbage quickly from one survivor space to the next.

Incremental Garbage Collector

While the scavenger only reclaims objects in New Space, the incremental garbage collector (IGC) also reclaims objects in Old Space. It does so incrementally, recycling dead objects by placing their headers and their bodies on the appropriate threaded free list. Note that the IGC does not compact object bodies or the object table and, thus, is ineffective against old space fragmentation.

The IGC can be made to stop if any kind of interrupt occurs, or it can be made to ignore all interrupts. In addition, you can specify the amount of work that you want the IGC to perform, both in terms of the number of objects scanned or the number of bytes scanned (the IGC will stop as soon as either condition is satisfied).

The IGC has five distinct phases of operation:

- Resting — the IGC is idle.
- Marking — the IGC is marking live objects.
- Nilling — the IGC is nilling the slots of WeakArrays whose referents have expired.

- Sweeping — the IGC is sweeping the object table, placing dead objects on the threaded free lists.
- Unmarking — the IGC is unmarking objects as a result of the mark phase being aborted, either at the user's request or because the IGC ran out of memory to hold its mark stack.

The typical order of operation is:

1 resting

2 marking

3 nilling

4 sweeping

5 resting

The unmarking phase is only entered if the mark phase is aborted, and it leaves the IGC in the resting phase when it is finished unmarking all objects.

With the exception of the nilling phase, each of the above phases is performed incrementally; that is, each can be interrupted without losing any of the work performed prior to the interruption. The IGC never performs more than one phase per invocation, unless the IGC notices that the marking phase has finished. When this occurs, nilling happens immediately and atomically to ensure that this is done while the object marks are current and consistent with the state of the image.

The provision for other phases (e.g., the marking phase) being able to do work incrementally permits clients to specify different workloads and different interrupt policies for the different phases. Consequently, clients will need to wrap their calls to the IGC in a loop if they want it to complete all of the phases. There is protocol for performing a complete IGC in the ObjectMemory class, the quickGC class method.

The object engine never invokes the IGC directly. Only Smalltalk code can run it. A typical memory policy might be to run the IGC in the idle loop, in low-space conditions, and periodically in order to keep up with the Old Space death rate. See Memory Policies for details.

Compacting Garbage Collector

The compacting garbage collector is a mark-and-sweep garbage collector that compacts both object data and object headers. This garbage collector marks and sweeps all of the memory that is managed by the object engine except for Perm Space, whose objects are treated

as roots for the purposes of this collector. This garbage collector is never invoked directly by the object engine, since the duration of its operation could be disruptive to the Smalltalk system.

Global Garbage Collector

The global garbage collector is a mark-and-sweep garbage collector that is identical to the compacting garbage collector except that it marks and sweeps all of the memory that is managed by the object engine, including Perm Space. This garbage collector is never invoked directly by the engine, since the duration of its operation could be disruptive to the Smalltalk system.

You might want to invoke the global garbage collector when you suspect that there are many garbage objects in Perm Space. This would reduce the size of the image file produced by a subsequent **Save As…**. The Global Garbage Collector would also reclaim the space occupied by garbage objects in Old Space, New Space that are only kept alive by references from garbage objects housed in Perm Space.

Both mark-and-sweep collectors use eden to store their mark stack. Although they will complete accurately if the mark stack overflows, they will require additional time to do so. Thus, if garbage collection fails due to a mark stack overflow, you may want to change the space allocated for eden at startup by sending a sizesAtStartup: message to ObjectMemory. For details, see Setting Object Engine Space Sizes.

In addition, the mark-and-sweep collectors use the Compiled Code Cache to store the list of ephemerons and weak objects that will require special handling at the end of the GC. If this list overflows, then not every ephemeron and weak object will be processed at once. Although typically this situation is extremely rare, you may want to increase the Compiled Code Cache size so no overflows occur.

Data Compactor

The system also has an Old Space data compactor. Because this facility does not try to compact the object table, or mark live objects, it runs considerably faster than either of the two mark-and-sweep garbage collectors. It should be invoked when Old Space data is overly fragmented.

For details using the data compactor, see Promoting Objects to Perm Space.

Relationship between space reclamation and the Compiled Code Cache

Because the mark-and-sweep garbage collectors use the Compiled Code Cache to hold the weak and ephemeron lists, the Compiled Code Cache is voided when invoking the garbage collector or the global garbage collector. However, the contents of the Compiled Code Cache are preserved by the incremental garbage collector, the data compactor, and the generational scavenger.

Some actions require voiding the Compiled Code Cache, and this is typically accomplished by collecting garbage. Make sure to use the mark-and-sweep collectors to do so, as opposed to the IGC. Also, note that the mark-and-sweep collectors are expensive. Even if the image has most of its old objects in Perm Space, thus giving a very light load to the non-global mark-and-sweep collector, performance may be greatly affected if the collector is invoked in a loop, for example:

```
100 timesRepeat:
   [      "drop reference to an object that should be collected"
      ObjectMemory garbageCollect
   ]
```

The reason for the performance penalty in this case may be due to the garbage collector's side effect of voiding the Compiled Code Cache.

Memory Policies

The object engine only supplies the low-level mechanisms for managing, allocating, deallocating, and garbage collecting object memory. It is up to the Smalltalk memory-management code to utilize these mechanisms in a judicious manner. This code belongs to the virtual image, and is accessible to the application developer.

The object engine provides an interface to the Smalltalk memory-management code via primitives, which are accessed from the class ObjectMemory and subclasses of AbstractMemoryPolicy. Developers wishing to access or modify the memory policies should use ObjectMemory's public protocol, or by creating a subclass of AbstractMemoryPolicy.

ObjectMemory

An instance of ObjectMemory represents a snapshot of object memory as it existed when that instance was created. The information contained in this object can be used to guide policy decisions for managing object

memory (see the subclasses of AbstractMemoryPolicy for examples of policies). Class ObjectMemory also contains protocol for manipulating the state of object memory.

In general, if you want to access the current state of object memory, you would create an instance of this class and then send messages to the instance. If, on the other hand, you want to directly manipulate the state of object memory (for example, to grow object memory, to compact object memory, or to reclaim dead objects that exist in object memory), you would do so by sending a message to the class itself.

Instances of ObjectMemory can also report some statistics such as the number of Stack Space spills, or the number of GC mark stack overflows. However, because the information contained in this class is implementation dependent and because it may vary from release to release, it is recommended that this information only be accessed directly by the low-level system code that implements the various memory policies or system profilers. Memory policy objects should provide an adequate set of public messages that will permit high-level application code to influence memory policy without resorting to implementation-dependent code.

Memory Policy Classes

Subclasses of AbstractMemoryPolicy implement the system's standard memory policy. In VisualWorks 7.7 and earlier, the memory policy was implemented by the class MemoryPolicy. (Subsequent to 7.7, this class is obsolete and will be removed in a future release.) Starting with 7.7.1, the default memory policy is implemented by the class LargeGrainMemoryPolicy. (MediumGrainMemoryPolicy is available as an alternate class.) In both cases, the policy is composed of two regimes for the object memory: one for growth and one for reclamation. The growth regime is in force when memory usage is below the growth regime upper bound, and the reclamation regime is in force whem memory usage is above it. LargeGrainMemoryPolicy favors using the IGC for reclamation purposes, setting the growth regime upper bound automatically at 60% of the memory upper bound.

In the growth regime, only the scavenger and incremental collectors are active. The object memory is allowed to grow freely upon demand, up to the growth regime upper bound, at which point the reclamation regime is entered. Under the reclamation regime, the object memory is not allowed to grow without a garbage collection of Old Space. Since garbage collection involves more overhead, overall performance is degraded somewhat during the reclamation regime. With

LargeGrainMemoryPolicy, adjusting the memory upper bound should produce a sensible growth regime upper bound for most circumstances. For details on how to adjust the growth regime upper bound, see Working with Memory Policies.

When the reclamation regime is in effect, memory will be allowed to grow if garbage collection fails to make space available up to the memory upper bound. If the application attempts to grow memory above the upper bound the system will enter an emergency low space condition.

The default memory policy response to an emergency low space condition is to interrupt the active user processes. For example, since Smalltalk stack frames are represented by Smalltalk objects, an infinite recursion can cause a low space condition, and the infinite recursion is interrupted once memory usage has grown up to the memory upper bound.

The memory upper bound is the main value that drives the behavior of all current memory policies. It is adjustable using the Memory Policy settings panel (see Working with Memory Policies). The default memory upper bound is 512 MB. You may need to adjust this value depending on your execution environment. For example, on machines with less RAM than the VisualWorks upper bound, the operating system will typically begin paging furiously and thrash badly once memory usage exceeds the amount of RAM. Consequently, if the memory upper bound is too high, an infinite recursion may take a long time to interrupt and a compacting GC will take a long time to complete. Note that this limit is enforced on a per-image basis. Hence, if your application relies on the collaboration of multiple images, the sum of their memory upper bound limits should not exceed the physical memory available to execute them.

Unfortunately, on many platforms supported by VisualWorks there are no operating system APIs to discover how much free RAM is available. Moreover, even if such an API were generally available, it is not possible to guess a suitable memory upper bound for every conceivable execution environment merely from the amount of available memory. Hence, VisualWorks does not automatically determine a memory upper bound. Therefore you must choose a suitable memory upper bound for your installation.

Free Space Upper Bounds

All memory policies grow Old Space and Fixed Space memory using the growOldSpaceBy: and growFixedSpaceBy: primitives which, if implemented, rely on the operating system's memory mapping facilities. Consequently, new segments can be returned to the operating system if the compacting garbage collectors can empty them. After the compacting garbage collectors run, memory policies use two free memory upper bounds to determine how much Old Space and Fixed Space space should be deallocated.

Note that if your application cyclically allocates large amounts of memory and releases it only to allocate the memory once more, you may find it efficient to increase the appropriate free memory upper bound, to reduce the amount of growth and shrinkage the system performs. For details on adjusting the free memory upper bounds, see Working with Memory Policies.

Default Memory Policy Behavior

Memory policy objects are given the opportunity to take action during the following circumstances:

- During the idle loop
- When the system runs low on space

In addition, memory policy objects are responsible for determining precisely what constitutes a low-space condition.

Memory policy objects take the following actions:

idle-loop action

Memory policy objects run the incremental garbage collector inside the idle loop, provided that the system has been moderately active since the last idle-loop garbage collection. The idle-loop action runs the incremental garbage collector in interruptible mode. In addition, the idle-loop action compacts object bodies if Old Space is too fragmented, and enforces the free memory upper bound.

low-space action

This action is how memory policies respond to true low-space conditions. If the system is biased toward growth, then the memory policy attempts to grow object memory. If, however, the system is not biased toward growth, or if object memory cannot be grown, then the memory policy tries various ways of reclaiming space. Failing that, the memory policy tries one last time to grow object

memory. Failing that, the memory policy attempts a global garbage collect to reclaim space. Failing that, the memory policy summons the low-space notifier.

The most interesting of these steps are the reclamation steps. Initially, memory policy objects will perform a full compacting garbage collector only if the free entries in the object table are consuming a significant percent of Old Space. If, on the other hand, a compacting garbage collector is not needed, the policy object will try to reclaim space by simply finishing the incremental garbage collector (if one is currently in progress). If that doesn't free up enough space, then the incremental garbage collector is run from start to finish without interruption. In addition, a data compaction is performed if Old Space is sufficiently fragmented. Nevertheless, note that, unlike the mark-and-sweep collectors, the IGC may fail if its mark stack overflows. Hence, if none of these operations frees up space, then the global garbage collector is invoked as a last attempt to reclaim space before stopping all user processes and bringing up the low space notifier.

Managing the Object Memory

Several different mechanisms are provided to give application developers precise control over the object memory and the policies for managing it:

- Promoting Objects to Perm Space
- Setting Object Engine Space Sizes
- Working with Memory Policies
- Creating a Custom Memory Policy

Promoting Objects to Perm Space

Moving Old Space objects into Perm Space (and thus improving the efficiency of garbage reclamation) is done by creating an image by choosing **File > Perm Save As...** in the VisualWorks Launcher window.

Creating an image in this way is similar to making a snapshot except that all of the objects that are currently in Old Space will be promoted to Perm Space when the new image is loaded back into memory at startup time. For details on these spaces in the object memory, see Perm Space and Old Space (above).

Alternately, you can cause all of the objects in Perm Space to be loaded into Old Space at startup time if you create an image using **File > Perm Undo As...** in the VisualWorks Launcher window.

Note that the current state of object memory is not changed by creating a new image using **Perm Save** or **Perm Undo**. In other words, only the newly created image will contain a modified Perm Space. For example, if you use **File > Perm Save As...** to create an image and later in that same session you create a normal snapshot on top of that image, Perm Space is unaffected.

To place your application code in Perm Space, follow these steps before deploying an image containing the application:

1 Create an image using the **File > Perm Save As...** command. Then choose **File > Exit VisualWorks...** and start the new image. All of the objects that were formerly in Old Space will be loaded into Perm Space, including the application code.

2 A number of transient objects will also inhabit Perm Space, such as those needed to display windows on the screen. To remove them, perform a global garbage collection.

3 Create a normal snapshot.

4 To make subsequent loads on the same platform even faster, you may want to load the new image back into memory and perform one last snapshot. This is useful because the global garbage collector compacts the objects in Perm Space, which forces the image loader to relocate these objects at startup time. By performing one extra snapshot, these objects will not need to be relocated on subsequent loads, when it is possible for the object engine to load them into their former locations.

Setting Object Engine Space Sizes

The default object engine memory space sizes are platform specific. Sizes for the following memory spaces can be adjusted at startup:

1 Eden (Object Creation Space)

2 Survivor Space

3 Large Space

4 Stack Space

5 Compiled Code Cache

6 Old Space Headroom

7 Fixed Space Headroom

To change any of these values, send sizesAtStartup: to the ObjectMemory class with an array specifying a multiplier for each space, then save and restart the image. The order of the array elements is as listed above.

Each multiplier must be a floating point betwee 0.0 and 1000.0. To get the requested memory size, the system applies the multiplier to the default size. A multiplier value of 1.0 yields the default size.

For example, to decrease Stack Space by 1/4 and increase Object Creation Space (a.k.a., Eden or New Space) by 1/2, while leaving the others at default sizes, send the message:

```
ObjectMemory sizesAtStartup: #(1.5 1.0 1.0 0.75 1.0 1.0 1.0)
```

This sets the size for the image at next startup. To make the new sizes take effect, save the image, exit VisualWorks and then restart the image.

These values are for illustration only. We recommend using the values currently set in the images we provide. Note the object engine may be started with a larger size if required to load the image. You can use the -m1 through -m7 command-line switches to override the sizes set by sizesAtStartup:.

Guidelines for Adjusting Memory Spaces

When adjusting memory spaces for most VisualWorks applications, you should first consider changing Eden (New Space). All new objects are created in Eden. If this space is too small, then objects can get tenured in Old Space too quickly, eventually causing IGC or GC overhead. On the other hand, if New Space is too big, the generational scavenger that scans New Space can begin to impact performance, because it will have to copy more objects from one survivor semi space to the next, and manage a larger remembered table. The “correct” size is generally a balance between processor speed, cache size, and application behavior. Generally, the default setting is acceptable.

Note that some performance problems related to new object creation are not best addressed by changing the size of New Space. For example, consider the following code:

```
| array |array := Array new: (10 bitShift: 20).
1 to: array size do: [:each | array at: each put: each asDouble]
```

The above code will require substantial code to run because the array will always be in the remembered table. Thus, on each scavenge, the scavenger will have to scan the entire array looking for objects that may

have been tenured into old space. Scanning 40 (or, in the case of 64-bit images, 80) megabytes of data per scavenge will greatly affect performance. For these cases, consider using segmented collection classes such as LargeArray:

```
Time millisecondsToRun:
   [      | array |
      array := Array new: (10 bitShift: 20).
      1 to: array size do: [:each | array at: each put: each asDouble]
   ]

Time millisecondsToRun:
   [      | array |
      array := LargeArray new: (10 bitShift: 20).
      1 to: array size do: [:each | array at: each put: each asDouble]
   ]
```

Using LargeArrays results in greater than 8 times faster execution (e.g., 3327 ms compared to 28283 ms).

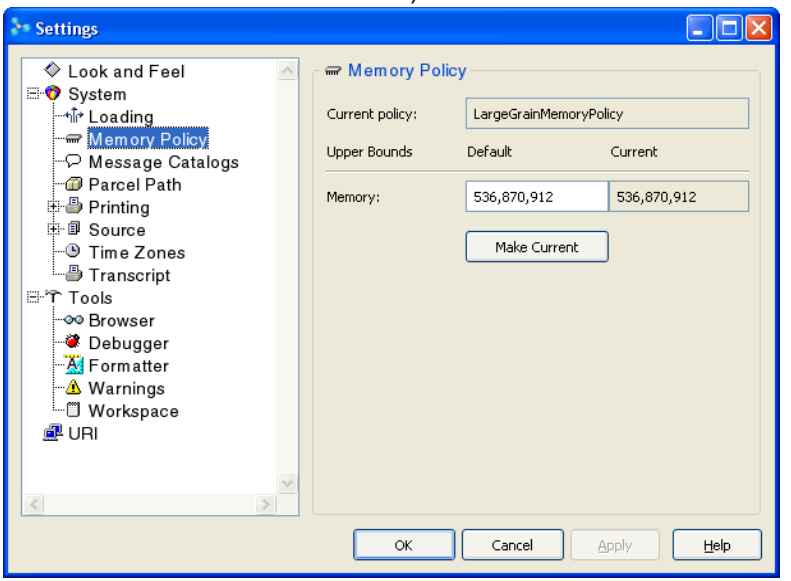
Because the VI manages Old Space, usually there is no need to manipulate it using the sizesAtStartup: method. Instead, you may control the size of Old Space configuring the memory policy's growth regime and memory upper bounds (for details on adjusting these values, see Working with Memory Policies). However, note that changing the size of New Space or Stack Space may require increasing the old space headroom to ensure that a worst-case scavenge will always succeed. Moreover, the old space headroom must be sufficient to allow VI initialization to the point that the VI memory policy is installed. If headroom is insufficient, you may experience a VM out of memory failure during image startup. In such cases, you can use the -m6 switch to manually increase the old space headroom. Typically, -m6 10000000 is enough. If the condition persists, you may have to increase the old space headroom multiplier permanently using sizesAtStartup:.

Working with Memory Policies

Because the default memory policy could be sub-optimal for a particular application, it is considered good practice to adjust the policy before deployment. The following discussion explains the various parameters that are available, and offers some guidelines for adjusting them to maximize the performance of your application.

For some applications, you may also want to create your own memory policy. For details, see Creating a Custom Memory Policy (below).

To examine the default memory policy settings, open the Settings Manager and select the **Memory Policy** page (choose **System > Settings** in the VisualWorks Launcher window):



Memory policies do not allow memory to grow above the memory upper bound. Instead, if this limit is reached, the currently active processes are interrupted by the low space notifier. Current memory policies derive all other necessary thresholds from the free memory upper bound.

Thresholds

Although the default memory policy is generally suitable for application development, it can almost always be optimized for deployment. Doing so requires an understanding of the basic three memory policy thresholds below:

- **Memory upper bound:** This is the limit past which the memory policy will refuse to grow old space. Because VisualWorks cannot sensibly guess a good value for this limit, the developer is responsible for providing a value that makes sense for the expected execution environment. In general terms, this value should produce an image that fits in physical memory. Moreover, if multiple images are running in the same machine, the sum of the images' memory upper bounds should be within the available physical memory. If the physical memory is not enough to hold all the images, excessive swap file thrashing may occur. The default value for the memory upper bound is 512MB.

- **Growth regime upper bound:** This is the limit past which the memory policy will try to collect garbage before growing memory. Generally, this limit should be enough to hold the expected application's object load. This limit defaults to 60% of the memory upper bound.
- **Free memory upper bound:** This is the limit of free memory past which the memory policy will return memory to the operating system (but note that this action may not reduce the VM's process footprint under Solaris because Solaris' implementation of free() reserves the previously allocated memory for future use by the process). This limit defaults to 20% of the memory upper bound, or twice the preferred growth increment (see below), whichever is greater.

The classes MediumGrainMemoryPolicy and LargeGrainMemoryPolicy are so named because they partition the memory upper bound with a given granularity. Thus, changing the memory upper bound automatically changes the values of the thresholds below to keep the same allocation granularity in effect. Although LargeGrainMemoryPolicy is the default memory policy, MediumGrainMemoryPolicy may be adequate for applications that favor reduced memory allocation. Both classes produce valid settings for memory upper bounds 64 MB and above. Users are encouraged to create new memory policy classes as necessary, as long as the thresholds and their associated constraints are thoroughly considered.

Interactions and Effects of Memory Policy Values

The various memory policy thresholds are set by the instance side initialization methods. Note that memory policies should not rely on the value of shared or class instance variables because changing these configuration values would affect the current memory policy with potentially undefined consequences. The method MediumGrainMemoryPolicy initialize method reads as follows.

```
initialize

    self updateMemoryStatus.
    self updateNumScavengesAsOfLastGC.
    self initializeHardMemoryBounds.
    self initializeAllocationGranularity.
    self initializeLowAndFreeSpace.
    self initializeFragmentation.
    self initializeIGC
```

The first line caches a new instance of ObjectMemory for later use. The second line initializes the number of scavenges as of the last GC operation. This value is held by the ObjectMemory instance cached in the line above. The scavenge count is used to determine whether enough allocations have occurred (measured by the scavenges of new space) to justify invoking the IGC in the idle loop. After this is done, a series of initialization steps occur.

```
initializeHardMemoryBounds

    self memoryUpperBound: self class defaultMemoryUpperBound.
    self contiguousSpaceSafetyMargin: self memoryStatus rtBytes
        + (self memoryStatus newBytesAvailableForStorage
            // self memoryStatus bytesPerOTE)
        + self memoryStatus largeBytes
        + self memoryStatus newBytesAvailableForStorage
        + self spaceSafetyMarginPadding.
    self availableSpaceSafetyMargin: self contiguousSpaceSafetyMargin
        + self memoryStatus oldRtBytes
        + self memoryStatus stackZoneFlushBytes
```

The above method records the memory upper bound from the default value set on the class side. Note that memory policies are validated for memory upper bounds of at least 64MB.

The available space safety margin is used to determine the hard low space limit. The hard low-space limit is the minimum available memory required for correct operation of the VM. The class ObjectMemory instructs the VM to signal a semaphore if the available memory falls below this limit. When the semaphore is signaled, the low space process tries to correct the situation by sending the message lowSpaceAction to the current memory policy. Note that this situation is potentially a memory emergency. Hence, if growth (if permitted by the memory upper bound), the IGC, and potentially a global GC are unable to bring the available memory above the hard low space limit, then the memory policy will tell ObjectMemory to suspend all user processes and bring up the low space notifier.

Since the VM will scavenge new space on its own, the memory policy must ensure there is enough memory for a worst-case scavenge, a scavenge in which every object in new space is tenured. The scavenger may also need to tenure new objects with bodies in large space, and the worst case scenario which results in tenuring an object as large as large space itself must be accounted for. In addition, note the care taken to allow the RT and the OldRT to grow as necessary during the scavenge. Moreover, since invoking the GC will also flush Stack Space,

there must be available memory to create enough Smalltalk context objects for a worst-case Stack Space flush. In addition, there is a space safety margin padding of 1MB for 32 bit images, and 2MB for 64 bit images. This margin is assumed enough to allow handle subsequent memory operations as needed. If the available space is not enough for a worst-case scavenge and a worst-case Stack Space flush, the VM will execute a controlled crash and output a message like

```
src/mman/mmScavenge.c 2375
Out of memory.
```

This is why it is critical for the memory policy to fulfil its responsibility to react to memory emergencies, and ensure the VM always has enough memory to perform its duties.

There may be cases in which the VI truly runs out of memory, such as when an infinite recursion is introduced in a system process. This is because, unlike user processes, system processes will not be interrupted by the low space notifier. You can usually find enough information to track down the offending process(es) by inducing VM failure after starting the image with the -xq VM switch. Upon running out of memory, the VM will dump the stack of all processes in a file called **`stack.txt`**. Infinite recursions can be easily found by examining the contents of this file.

The contiguous space safety margin is similar to the available space safety margin, but in this case it represents the minimum amount of memory under which the VM runs the risk of crashing. In this case, it depends on the size of the RTs (which are assumed to grow to twice their current size in a worst-case scenario). It also depends on the size of new space, because a worst-case scavenge may tenure a pointer object such as an Array or OrderedCollection as large as new space. Similarly, there must be enough contiguous space to tenure a large space object as large as large space itself. In addition, there must be enough contiguous space to grow the RT as needed (the OldRT does not need as much contiguous space to grow since it is segmented). Finally, this value is also padded by 1MB on 32 bit images, and 2MB on 64 bit images.

```
initializeAllocationGranularity

    self growthRegimeUpperBound:
        ((self targetGrowthRegimeUpperBound max: (48 * self oopSizeFactor
            bitShift: 20))
            min: self memoryUpperBound * 3 // 4).
    self preferredGrowthIncrement:
        (self targetPreferredGrowthIncrement max:
```

```
        (self availableSpaceSafetyMargin max: self
            contiguousSpaceSafetyMargin)).
    self preferredFixedGrowthIncrement:
        (self preferredGrowthIncrement // 8 max: 1048576 * self
            oopSizeFactor).
    self growthRetryDecrement: self preferredGrowthIncrement // 96
```

From the memory upper bound, the growth regime upper bound is set to a fraction of the memory upper bound. The default memory policy sets this boundary at 60% (or three-fifths) of the memory upper bound. Thus, the LargeGrainMemoryPolicy will prefer image growth to space reclamation until the image reaches 60% of the memory upper bound.

The preferred growth increment is derived as a fraction of the memory upper bound as well. However, since this value cannot be smaller than either the available space safety margin or the contiguous space safety margin, the resulting fraction of the memory upper bound may be larger than the target size.

The preferred growth increment dictates the size of additional old space segments. In general terms, numerous, comparatively small old space segments (e.g., a 500MB image with a 1MB preferred growth increment may result in over 400 old space segments) will result in bad compacting GC performance. In addition, particularly for 32-bit images, the VM may be unlucky and allocate a multitude of small segments such that the OS will not find enough contiguous address space for a large old segment needed to house a large pointer object. Thus, small old segments may induce allocation failures even though it may appear as if there is enough available memory. Comparatively larger old segments result in significantly faster compacting GC performance and less risk of memory fragmentation. Their drawback is that it will be more likely for them to have some data and, since old segments can only be deallocated when they are empty, the memory policy may not be able to return memory to the operating system. The default memory policy assumes it is preferable to use a larger value for the preferred growth increment.

If an attempt to grow memory by the preferred growth increment fails, in some cases the memory policy will attempt to grow memory by less than the preferred growth increment. The growth retry decrement controls how much to give up by in each growth attempt. Generally speaking, it should not be too small compared to the preferred growth increment.

The preferred fixed space growth increment is the Fixed Space analog to the Old Space preferred growth increment.

```
initializeLowAndFreeSpace

    self freeMemoryUpperBound: (self memoryUpperBound // 5 max:
        self preferredGrowthIncrement * 2).
    self freeFixedMemoryUpperBound: self preferredGrowthIncrement // 4
```

The free memory upper bound specifies the maximum allowed amount of free memory in old space. Its value is initially set to 20% (or one-fifth) of the memory upper bound. However, to prevent situations in which applications cause a cycle of allocation and deallocation above the growth regime upper bound due to the memory policy's eagerness to give memory back to the OS, it should be at least twice as large as the preferred growth increment.

The free fixed memory upper bound specifies how much free fixed memory will be allowed before the memory policy tries to deallocate fixed space segments. Because applications will generally use much less fixed space data than old space data, its value is set to 25% (or one-fourth) of the preferred growth increment.

```
initializeFragmentation

    self threadedDataIncrement: 25000 * self oopSizeFactor.
    self idleMaxPercentOfThreadedData: 50.
    self idleMaxPercentOfThreadedOTEs: 7.
    self maxPercentOfThreadedData: 75.
    self maxPercentOfThreadedOTEs: 10.
    self maxProbesPerAttempt: 10
```

The idle max percent of threaded data specifies the percentage of potentially fragmented old space past which the memory policy will invoke the data compactor in the idle loop. Note that the theaded data refers to old space fragments that were once occupied by live objects, but that have since not been reused. The idle max percent of threaded object table entries (OTEs) is the limit of object table entries past which old space is deemed to be too fragmented. Exceeding this limit will result in the memory policy invoking the data compactor.

The maximum percent of threaded data and OTE limits are as above, except they are the absolute limits past which the memory policy will assume data is too fragmented. Consequently, when fragmentation goes past these limits, the memory policy will invoke the data compactor.

```
initializeIGC

    self allowIncGC: true.
    self idleLoopAllocationThreshold: self preferredGrowthIncrement * 3.
    self incrementalAllocationThreshold:
        (self preferredGrowthIncrement // 3 min: (4097152 * self
            oopSizeFactor max: self preferredGrowthIncrement // 10 + 1)).
    self incMaxPauseMicroseconds: 20000.
    self incMaxMarkQuota: 100000.
    self incMaxSweepQuota: 500000.
    self incMaxUnmarkQuota: 500000
```

Memory policies support enabling and disabling the IGC on demand using the message allowIncGC:. By default, the IGC is enabled. Applications should consider temporarily turning off the IGC for additional performance when a very large amount of non-garbage objects is about to be created.

The idle loop allocation threshold is the amount of allocated memory, measured in new space scavenges, that justifies running the IGC in the idle loop. Because its value depends on how much old space may have been potentially used due to the scavenger tenuring objects, it is set to a multiple of the preferred growth increment. The default is three times the preferred growth increment. If this value is too small, then the IGC will be run interruptably in the idle loop even though the potential savings may be too small.

The incremental allocation threshold also controls whether the IGC runs in response to continued old space allocations. Its value is used to set the soft low space limit. ObjectMemory sets the low space semaphore to the soft low space limit or the hard low space limit, whichever is greatest. If the low space action is invoked but there is no memory emergency, then the memory policy will invoke the IGC in an attempt to free garbage that may have made it past the new space generational scavenger. By default, this value is set to a fraction of the preferred growth increment. If this value is too small, then the IGC will run uninterruptably even though the potential savings may not be worth the effort.

Current memory policies actively measure IGC performance, and will target IGC work quotas so that uninterruptible incremental steps do not require more than incMaxPauseMicroseconds. The work quotas provided in the above method are merely initial values. Memory policies will automatically adjust the quotas, responding to changing conditions on the fly.

The class LargeGrainMemoryPolicy differs from MediumGrainMemoryPolicy in a couple target sizes. Hence, the above discussion applies to LargeGrainMemoryPolicy without loss of generality.

Setting Memory Policy Values

The best way to set these parameters is to first measure your application's memory usage. To get an idea of memory usage, we recommend using the MemoryMonitor, in **`contributed/`**, particularly because of its ability to write CSV logs headlessly. When measuring memory usage, it s important to measure the size of the object memory at points of high load. If measurements show that no memory is reclaimed and the application is still performing work, the current memory usage could be a reasonable high water mark.

Use the results to set the memory policy parameters. The first and most obvious adjustments can be made to the growth regime and memory upper bounds. The growth regime upper bound should be slightly higher than the peak usage. This bound must accomodate the maximum expected working set for your application. The growth regime upper bound should be set to keep the time spent in garbage collection to an acceptably low interval. Attempts to grow the object memory beyond this limit cause VisualWorks to garbage collect before asking for more heap memory from the host OS. If the limit is too low, the application spends too much time garbage collecting. If it's too high, the application wastes memory by not garbage collecting. This value must be adjusted considering that the IGC should be given a chance to collect garbage in the background, thus avoiding the interruptions caused by the mark-and-sweep collectors.

The memory upper bound should be set to the application's maximum footprint plus a safety margin. You should keep in mind that since memory growth is not allowed beyond the upper bound, computations will be interrupted with a low space error if memory usage ever reaches the upper bound. The safety margin should be enough to meaningfully react to this error.

The free memory upper bound limit may need to be raised so that the application doesn't release as much free memory after a garbage collection. While in principle it sounds good to have the application return memory to the host operating system after garbage collection, in practice this may degrade performance. Note that, under Solaris, the VM will not be able to return memory to the operating system due to the behavior of the free() function.

As described about, the VisualWorks memory policy contains several adjustable parameters, which are not shown in the Settings Manager. To adjust these, you must create your own custom policy class.

Creating a Custom Memory Policy

In some cases, you can improve performance by using a custom memory policy. You may change any aspect of the policy, including both the constants and algorithms used to manage the object memory, as long as the basic responsibilities of a memory policy are met.

The basic steps to create a new policy are:

1 Define a new memory policy class (e.g., MyMemoryPolicy). This class should inherit from AbstractMemoryPolicy (or one of its subclasses), and provide a concrete implementation for the instance protocol defined in terms of subclassResponsibility in AbstractMemoryPolicy. Note that your memory policy class must implement install on the class side, as implemented in AbstractMemoryPolicy.

2 Make sure to implement the appropriate initialization methods and set the appropriate variables. For example,

```
targetGrowthRegimeUpperBound

    ^self memoryUpperBound * 4 // 5
```

To make sure the resulting memory policy is consistent, use the MemoryPolicyChecker test parcel. (You may need to add a test case for your memory policy; use LargeGrainMemoryPolicyChecker as a template). For additional tuning help, you may want to use the MemoryPolicyTuner test parcel.

3 Install the new policy:

```
MyMemoryPolicy install
```

You may also want to use the MemoryPolicyStressTest parcel to ensure the new memory policy can handle stressful memory management situations.

In this example, we have created a new policy that uses a different growth regime upper bound. This is the amount past which the memory policy will collect garbage before growing old space.

There are additional situations that may require a customized memory policy. For instance, it may be convenient to let go of accessory application data such as caches when memory is running low and a large object allocation fails.

If allocating a Smalltalk object with the message new fails, chances are there is a memory emergency in progress already because objects created by new will be far smaller than the hardLowSpaceLimit. In such cases, you should refine the actions performed by the message lowSpaceAction to release unnecessary data. For example, the application could refine lowSpaceReclamation to reinitialize caches if garbage collection is unable to make progress.

When allocating a Smalltalk object with the message new: fails, the memory policy receives a request to make room for a certain object type before the primitive is retried for the last time. The application could refine the memory policy to dump accessory information in an attempt to make space if the attempt to make space fails.

Applications should be careful not to simply grow more memory when a primitive fails to allocate OS resources as opposed to Smalltalk objects. For example, if primitive 900 fails because it cannot obtain a handle, the correct action would be to perform a GC so that potentially unused handles can be collected and finalized. Once this is done, the primitive can be retried one last time in an attempt to obtain the necessary OS handle.

For a complete description of the memory policy API, see the class comment in AbstractMemoryPolicy.

MemoryPolicy Strategies

A custom memory policy might perform application-specific actions, flushing application caches, making policy decisions about process allocation, and so forth. For example, the VisualWorks Application Server uses VisualWave.ServerMemoryPolicy to assess system load, expire web existing sessions, and refuse new connections. Another example of a custom policy would be one that maintains a pre-specified range of memory usage. That is, if the available memory drops below a lower threshold, the policy enters the growth regime, and if it reaches an upper threshold, it enters the reclamation regime.

Preparing for Deployment

The following steps are recommended for deploying a VisualWorks image:

1 Load application code.

2 Prepare to create a new image, promoting Old Space objects into Perm Space (in the Launcher window, select **File > Perm Save As…**).

3 Run the Global Garbage Collector (in the Launcher, select **System > Collect All Garbage**).

4 Create a snapshot (in the Launcher, select **File > Save As…**).

5 Adjust the memory policy paramaters as necessary (for details, see Working with Memory Policies, above).

6 Run the Global Garbage Collector (in the Launcher, select **System > Collect All Garbage**).

7 Save the image in a ready-to-run state.