

Controlling VisualWorks' NewSpace

A space for everything in its space

John M. McIntosh

Recent discoveries made while building a system that worked as a server instead of as a typical GUI application have led me to write this article. During the testing phase, I was surprised to see that the image grew from a starting size of about 8 MB to roughly 18 MB before it stopped requesting memory from the operating system. This behavior led to the questions: Why does it grow? Why does it stop growing? A manual garbage collection usually returned the response that 8 MB of memory had been freed. Most puzzling. Although the image didn't grow continuously, I thought I had understood how it used memory, and now was forced to take a closer look at the problem. This article is the first of a series explaining how Garbage Collection (GC) is done in VisualWorks.

My starting point was extensive reading of the ParcPlace-Digitalk manuals and examination of the image. I ran across the MemoryPolicy class comment, which states: "A typical memory policy might be to run the Incremental Garbage Collector (IGC) in the idle loop, in low-space conditions, and periodically in order to keep up with the OldSpace death rate." Light bulbs! My application does not fit the regular pattern of GUI applications! Idle times are a rarer event for server applications, since they do not enjoy human interface pauses. Instead, they might service many users, and always have a high-activity level. Armed with my apparent lack of memory-management knowledge, I embarked on a journey to discover exactly how Smalltalk deals with memory beyond the Scavenge.

GARBAGE COLLECTION

To understand how the MemoryPolicy class interacts with the image, one first needs to step back and understand how garbage collection works. Some background information can be found in Kent Beck's "Garbage Collection Revealed," *The Smalltalk Report*, Vol. 4, No. 5, Feb. 1995, which talks about VisualSmalltalk, and gives the reader a

detailed introduction into GC theories. More information can be found on the Web at <ftp://ftp.netcom.com/pub/hb/hbaker/home.html>, which contains access to a number of papers on GC theories. Hewitt and Lieberman's paper "Lifetimes of Objects,"¹ and David Ungar's classic paper "High Performance Smalltalk Systems,"² also lay groundwork for the GC logic used by current commercial Smalltalk systems. These papers point to a key discovery: namely, that most objects are short lived. In fact, 80 to 98

percent of objects die shortly after birth. The survivors generally live for a long time.

With this observation in mind, a typical Smalltalk system divides memory into two areas: NewSpace for object creation and OldSpace for long-lived objects. The objective of a memory-management system is to aggressively do GC work on a

small area of memory. This activity can be done within the pause time between keyboard keystrokes, so the impact on the user isn't noticed. From time to time, checking all objects in a multimegabyte image for survivors is done, by using an incremental GC that runs when the system isn't doing more important work. Running short on memory will trigger more drastic measures to find and remove dead objects, before declaring a critical memory shortage. An application that doesn't match well with the expected behavior will suffer from pauses, and possibly use excessive swap space as the OldSpace GC logic attempts to fix a problem that might be solvable within the domain of the NewSpace GC logic.

NEWSPACE GC LOGIC

NewSpace is really three areas, with a tenuring extension³ to make four. These areas are used as a base for the garbage-collection algorithm. 'Eden' is where objects are first allocated, or born. The next two areas are Survivor spaces. One contains live objects, while the other is empty, being used during an event called "The Scavenge." A

The objective of a memory-management system is to aggressively do GC work on a small area of memory.

fourth space, 'LargeSpace', is used to handle large objects as a modification to the original algorithm, and attempts to reduce the movement of large byte objects between Survivor spaces. Strings that exceed roughly 1K are created in LargeSpace with a link to Eden.

In a small VW 2.5 Windows NT image, the sizes of the various spaces are:

Eden	204,800 bytes
Survivor Space A	40,960 bytes
Survivor Space B	40,960 bytes
Large Space	204,800 bytes

THE SCAVENGE

When Eden fills to the Eden byte-used threshold, the VirtualMachine (VM) invokes an event called a scavenge. The verb "to scavenge" is aptly defined in the *Webster's New Collegiate Dictionary* as: "to remove (as dirt or refuse) from an area, or to salvage from discarded or refuse material." During a scavenge, the VM locates all objects in Eden and the active SurvivorSpace reachable by the systems roots, and copies them into the empty SurvivorSpace. Once the scavenge examines Eden and the original active SurvivorSpace, those spaces now only contain dead objects, and are deemed empty. Memory allocation starts again, with objects being placed into Eden.

As you can see, survivor objects are shuffled between the two SurvivorSpaces on each scavenge, with new objects being added from Eden. Objects that die in

SurvivorSpace are not copied during the scavenge, and overall growth is based on how good or bad your application is in creating and holding new objects. You will notice that 40K or so of memory isn't very big, so once the number of bytes in SurvivorSpace reaches the defined threshold, the scavenger will tenure objects from SurvivorSpace to OldSpace until there is room in the SurvivorSpace.

OLDSPACE

OldSpace is many memory segments that combine to form a virtual chunk of contiguous memory. This leads to the external behavior shown to the hosting operating system. A VW image will grow by chunks; once the memory is allocated from the hosting operating system, it is not returned. Images never shrink, they just grow. Great, but with all that activity hidden in the VM, can NewSpace garbage collection be controlled? Certainly. In fact, you need to control garbage collection to solve a problem known as the 'Early Tenuring Issue'—the result of an application holding objects for a few hundred milliseconds too long.

When this happens, objects are tenured into OldSpace too early and promptly die, thus defeating the generation scavenging logic. This shifts NewSpace GC work to the OldSpace GC logic. To show how this happens, let us create an object that will artificially hold items and cause the early tenuring problem. We will then alter the size of SurvivorSpace to observe how it will affect performance. Source code will follow this article. To alter the size of NewSpace, you must use `ObjectMemory class>>sizesAtStartUp:`

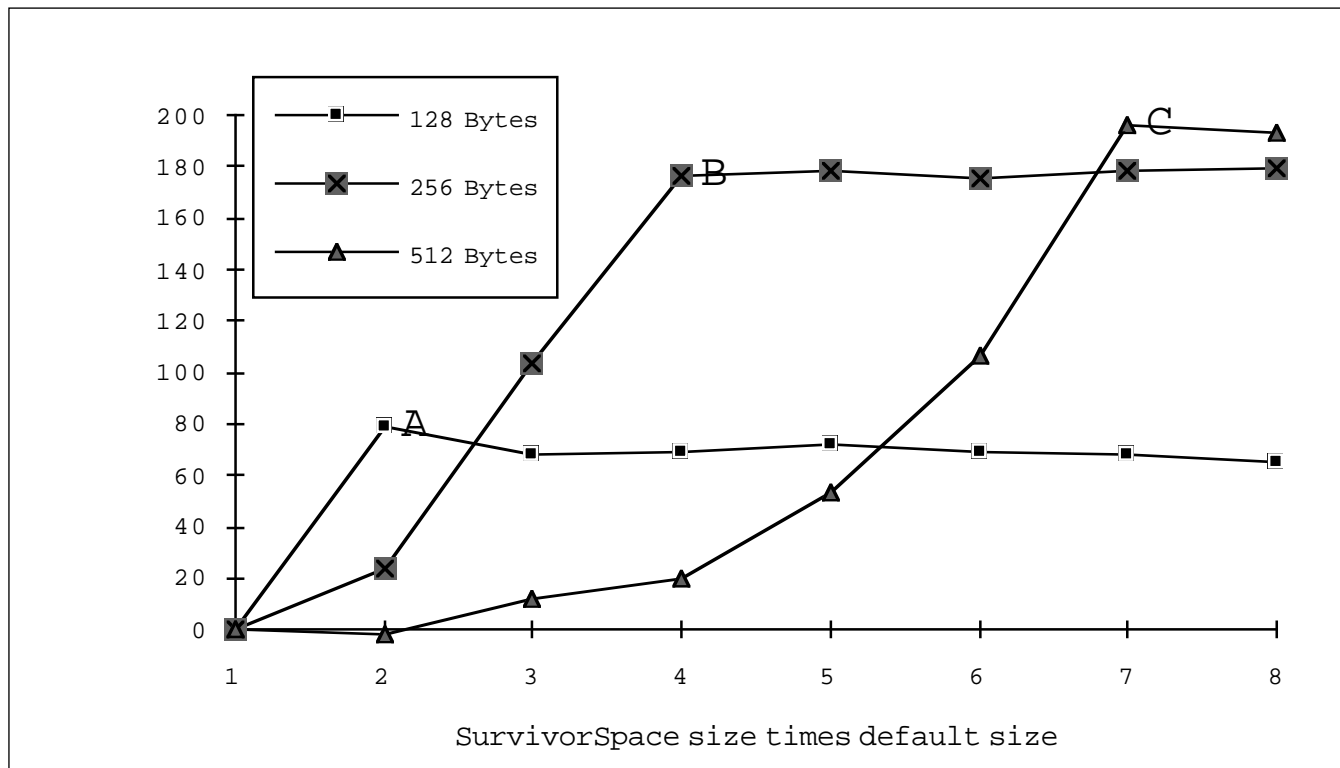


Figure 1. This figure shows restricted image growth, and allocation-rate improvements against default NewSpace size of 40K.

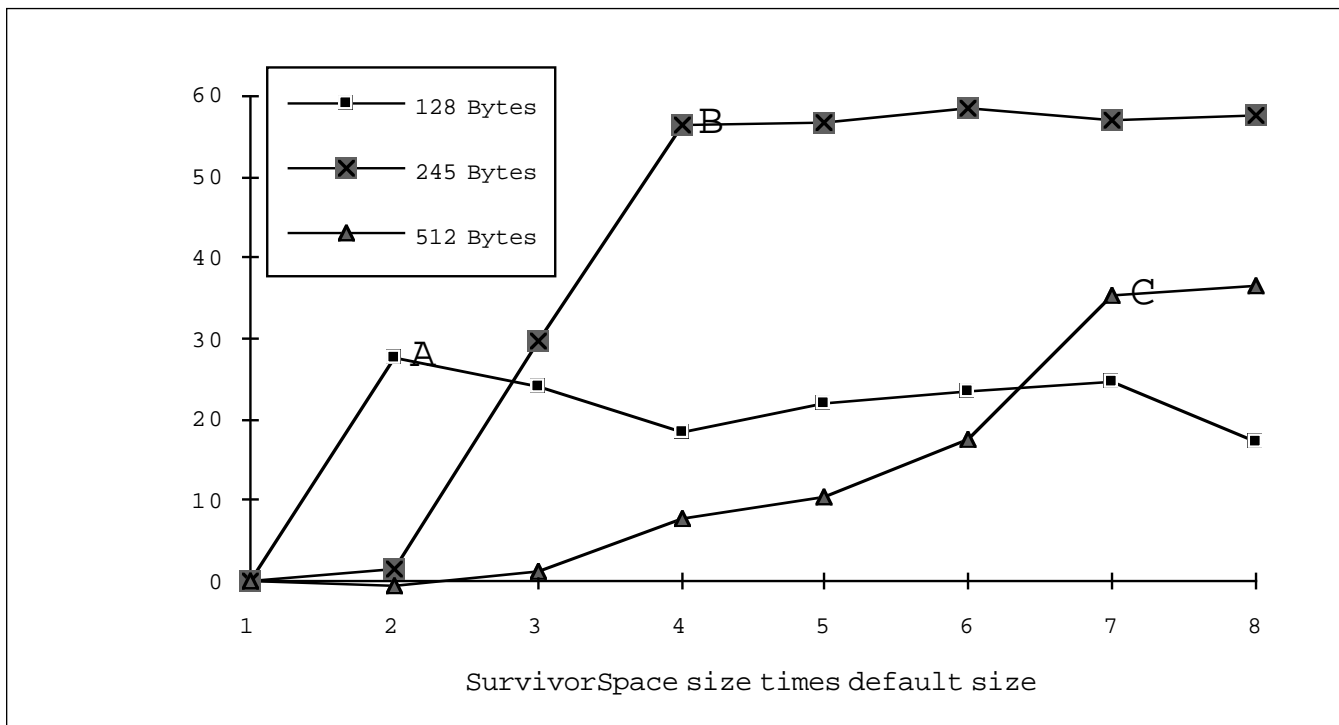


Figure 2. This figure depicts unrestricted memory growth.

and `ObjectMemory class>>thresholds`: to change the default sizes and thresholds, which were chosen by ParcPlace. The `sizesAtStartup`: method allows me to alter the amount of memory VW allocations for each memory area at startup. The `thresholds`: method dictates the thresholds for Eden, Survivor, and LargeSpace.

Two conditions were tested. Since memory is not free, I limited the amount of memory that the image could allocate for one of the tests. The other had full freedom to extend the image. For both tests, I altered the size of NewSpace from two times the default size up to eight

OldSpace is many memory segments that combine to form a virtual chunk of contiguous memory.

times the default size. These changes allowed us to observe the effects of a survivor space that varied from 80K to 320K.

The `EarlyTenureTest` object allocates an Array of 500 elements. For a certain number of seconds, a loop is performed, where a new String object of a given size is allocated and placed in the Array, starting at element one. The index is incremented, and a new string is allocated into the next element. When the last element of the Array is reached, it starts again at element one. If the SurvivorSpace isn't large enough to contain the full working set of the

Array and its components, some of these strings will be tenured into OldSpace.

To show how the image behaves under different conditions, and how OldSpace GC really impacts performance, we first restrict the image's size to 8MB.

In Figure 1, we see allocation-rate improvements against the default NewSpace size of 40K. `EarlyTenureTest` instances are created using a string size of 128, 256, and 512 bytes. These instances require a working set size of at least 64K, then 128K, and finally 256K bytes. On reviewing Figure 1, it is clear that the 128-byte allocation rate improves by about 80%, when we go to a Survivor size of 2x (see point A). For the 256-byte instance, the SurvivorSpace needs to go to 4x before the allocation peak (see point B). Finally, for the 512-byte instance, we peak at a SurvivorSpace size of 7x, with an improvement of almost 200% to the allocation rate (see point C). The impressive improvement for the 512-byte instance happens when we avoid expensive OldSpace GC work. In all three cases, there is a net improvement in the overall work done by the application. Figure 2 shows what happens if memory growth is not restricted.

Although the increase in memory-allocation rates is not as impressive as in our first case, image growth is affected. Using the default size, the image grows from 8MB to 13MB. Changing the size to 7x keeps the image at 8MB, and improves allocation performance from 25% to almost 60%. Again, points A, B, and C show the plateaus where we get the best allocation improvement for the 128-, 256-, and 512-byte instances. Image growth may be free, but allowing it means managing a larger OldSpace, and this can ultimately impact performance.

For both free-growth and restricted-growth situations,

the ending memory-allocation rate is roughly the same, once we reach seven times the default SurvivorSpace size. The image in both cases stabilizes at a dynamic footprint of about 8MB. A larger SurvivorSpace improves memory allocation throughput and reduces image growth. More is better—a “win/win” situation.

In many cases, changing the size of SurvivorSpace means tenuring problems can be traded for slightly

A larger SurvivorSpace improves memory allocation throughput and reduces image growth.

more time spent on NewSpace GC work. Many of the thresholds decided by ParcPlace-Digital for VW date from 1990, and CPU performance, have greatly increased since then. One can easily increase the amount of memory that the scavenger needs to examine, without noticing any effects on response time; and as our examples show, you can improve your application’s performance by 200%!

Of course, your application may not have a small working set. Even so, some tests are worth doing. Consider altering your SurvivorSpace allocation by a factor of 10x, and observe the final dynamic memory footprint and time needed to complete a certain task.

This article addresses only NewSpace GC work. In an upcoming issue, I will discuss how memory is allocated, and what happens if you don’t have sufficient memory on hand when you ask for another MB (or two) of that elusive resource.

SUPPORTING CODE

From VisualWorks(R), Release 2.5 on September 26, 1995:

```
Object subclass: #EarlyTenureTest
  instanceVariableNames: 'holdTooLong counter
allocationSize trackAllocations logStream canStop
waitSync '
  classVariableNames: "
poolDictionaries: "
  category: 'JMM-Memory-Paper'!
EarlyTenureTest comment:
```

©1996 John M. McIntosh, All Rights Reserved.
johnmci@ibm.net.

An object that creates the EarlyTenure problem so we can examine NewSpace behavior:

Instance Variables:

```
holdTooLong      <Array> holder for strings of
                  size allocationSize
```

```
counter          <Integer> iterates over the array to place
                  elements
allocationSize   <Integer> holds current allocation size
                  for Strings
trackAllocations <Integer> hold total number of alloca
                  tions
logStream        <Stream> log of information about
                  test cycle
canStop          <Boolean> true when I can stop
waitSync        <Semaphore> used to sync workand
                  result log
```

Note: to test this you must invoke the sizesAtStartup:, then quit and save your image. After restart of the image, invoke thresholds: to reset the NewSpace memory threshold. You may need to adjust sizesAtStartup: if your installation has already altered some of the other memory space sizes.

```
ObjectMemory sizesAtStartup: #(1.0 7.0 1.0 1.0 1.0 1.0).
ObjectMemory thresholds: #(0.96 0.95 0.90)!
```

```
!EarlyTenureTest methodsFor: 'actions'!
```

```
createElement
  self holdTooLong at: self incrementCounter
  put: (String new: self allocationSize)!
```

```
incrementCounter
  self trackAllocations: self trackAllocations + 1.
  ^counter := counter >= self defaultCounter
  ifTrue: [1]
  ifFalse: [counter + 1]!
```

```
runForThisManySeconds: aNumber
  "Fork the delay timer, fork the work, when done return
the logStream contents"
```

```
self forkTimer: aNumber.
self forkAllocation.
self waitSync wait.
^self logStream contents!
```

```
writeSize
  "Print the time, memory footprint, scavenges and total
allocations for our records"
```

```
self logStream nextPutAll: Time now printString;
space;
nextPutAll: ObjectMemory
dynamicallyAllocatedFootprint printString;
space;
nextPutAll: ObjectMemory current numScavenges
printString;
```

```
space;
nextPutAll: trackAllocations printString;
cr! !
```

```
!EarlyTenureTest methodsFor: 'defaults'!
```

```
defaultCounter
^500!
```

```
defaultPriority
^Processor userBackgroundPriority!
```

```
defaultSize
"Do not change over 1000 bytes "
^512! !
```

```
!EarlyTenureTest methodsFor: 'accessing'!
```

```
allocationSize
^allocationSize isNil
ifTrue: [allocationSize := self defaultSize]
ifFalse: [allocationSize]!
```

```
allocationSize: aNumber
allocationSize := aNumber!
```

```
canStop
^canStop!
```

```
canStop: aFlag
canStop := aFlag!
```

```
holdTooLong
^holdTooLong!
```

```
holdTooLong: anArray
holdTooLong := anArray!
```

```
logStream
^logStream!
```

```
logStream: aStream
logStream := aStream!
```

```
trackAllocations
^trackAllocations!
```

```
trackAllocations: aNumber
trackAllocations := aNumber!
```

```
waitSync
^waitSync!
```

```
waitSync: aSync
waitSync := aSync! !
```

```
!EarlyTenureTest methodsFor: 'initialize-release'!
```

```
initialize
```

```
holdTooLong := Array new: self defaultCounter.
counter := 0.
logStream := WriteStream on: (String new: 1024).
canStop := false.
trackAllocations := 0.
waitSync := Semaphore new! !
```

```
!EarlyTenureTest methodsFor: 'forks'!
```

```
forkAllocation
```

```
[self writeSize.
[self canStop] whileFalse: [self createElement].
self writeSize.
self waitSync signal]
forkAt: self defaultPriority!
```

```
forkTimer: aNumber
```

```
[(Delay forSeconds: aNumber) wait.
self canStop: true]
forkAt: self defaultPriority + 1.!!
```


```
"-----"!
```

```
EarlyTenureTest class
instanceVariableNames: ""!
```

```
!EarlyTenureTest class methodsFor: 'instance creation'!
```

```
new
^super new initialize! !
```

```
!EarlyTenureTest class methodsFor: 'Example'!
```

```
example
^self new allocationSize: 128; runForThisManySeconds:
15.!! 
```

References

- 1) Lieberman, H., and Hewitt, C., "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *CACM* 26,6, June 1983, pp 419-429.
- 2) Ungar, D. M., "Generation Scavenging: A Non-disruptive High-Performance Storage Reclamation Algorithm," *Proceedings of the {ACM SIGSOFT/SIGPLAN} Software Engineering Symposium on Practical Software Development Environments*, June 1984, pp 157-167.
- 3) Ungar, D. M., and Jackson, E., "An Adaptive Tenuring Policy for Generation Scavengers," *ACM Transactions on Programming Languages and Systems*, Vol. 14, No 1, January 1992, pp 1-27.

John McIntosh is an independent Smalltalk consultant. After eight years of building client/server applications, he discovered Smalltalk. It was love at first sight! He is currently building Web applications for a Silicon Valley company. He can be contacted by phone at 800-477-2659 or by email at johnmci@ibm.net.