GemStone Garbage Collection in a Production Environment

Norman R. Green
September 29, 1999
Revised April 23, 2022 by Bob Bretl,
to reflect the changes from the 32 bit to the 64 bit architecture

This paper describes the internal behavior of GemStone Garbage Collection, and how to analyze performance and configure your GemStone system to collect garbage efficiently and with the least impact on a production system.

This details of the internal workings of GemStone are subject to change in new versions of GemStone, and the recommendations are based on "typical" configurations. Tuning inherently involves observing the effect of an adjustment on the behavior, for a specific application, hardware, and GemStone version.

The information in this document is intended for experienced GemStone64 Database Administrators, and is written with the assumption of detailed knowledge of GemStone configuration and administration.

Important Definitions

- **Root Set –** A reserved set of objects that serve as an anchor for all objects in a GemStone repository.
- **Live Object –** an object that is referenced by one or more objects connected to the root set.
- **Dead Object –** an object that is not referenced by any live object. Other dead objects may reference a dead object. Since dead objects are not needed, both the space and the identifier (OOP) used by the object may be reclaimed and reused.
- **Possible Dead Object –** an object which appears to be a dead object but has not yet been finalized. A possible dead object is "promoted" to a dead object status after finalization.
- **Shadow Object –** an old copy of a live object with the same object identifier. The space used by shadow objects may be reclaimed but the identifier may not.
- **Cache Frame –** a slot in the shared page cache which may hold a page.
- **Free Frame –** a cache frame that is not in use and is on the list of free frames maintained for the shared cache.
- **Free Frame Limit –** a gem configuration setting which governs how many frames that gems may consume from the free frame list. When the number of frames remaining on the list is below this value, gems may no longer remove frames from the list. Instead, gems must scan the shared cache for a frame. The default value of the free frame limit for all gems except the GcGem is 10% of the total frames in the shared cache. For the GcGem, the default is 5%.
- **OOP –** Object Oriented Pointer. Each object is assigned a unique numeric identifier by the system.
- **OOP Number** – a sequentially allocated 40 bit number representing a persistent object that is allocated an entry in the object table
- **POM OOP –** a Persistent Object Memory OOP. This is a 64 bit value used to represent persistent objects. The lower 8 bits indicate what kind of object it represents, e.g. SmallInterger, Double, etc.. Objects that have entries in the object table are represented as (OOP Number << 8) + 1.
- **OOP high water mark –** the value of the highest OOP Number ever allocated in the system.
- **Object Table (OT)-** an internal GemStone structure that maps Oops to a disk location.
- **Object Table Page –** a disk page which contains object table information.
- **Data Page –** a disk page contains data objects. Most pages in the repository are data pages.
- **markForCollection (MFC) –** the GemStone method that invokes the global garbage collection process. Execution of the markForCollection method does not in itself remove garbage objects from the repository. It only computes the list objects eligible for removal.
- **Write Set –** a set of objects that a gem has modified. The write set is used during commit processing.
- **Write Set Union –** the union of all write sets from a collection of commits. The stone will keep track of the write set union while garbage collection is in progress so it can be used during the finalization stages.

GemStone64 Garbage Collection Overview

The focus of this document is on persistent object garbage collection. There are also garbage collection operations performed in individual gem processes that clean up unused objects in the Temporary Object Cache (TOC), but these are not covered in this paper.

There are three basic garbage collection operations performed on the persistent objects in the Gemstone Repository:

- markForCollection – the primary method for detecting garbage in the repository
- Epoch GC – (garbage collection over a specific time interval)
- Symbol GC – (garbage collection of symbols allocated in the system)

The complete garbage collection cycle has 6 stages:

1. Detection of possible garbage objects (possible dead)
2. Voting
3. AdminGem: Write Set Union Sweep
4. Promotion to dead
5. Reclamation
6. Return Oops and Pages to free pools

The detection of the dead objects occurs in either a markForCollection or epochGc operation.

Execution of the markForCollection method and epochGc are mutually exclusive. The system enforces this by acquiring a garbage collection lock (gcLock) which only allows one operation at a time.

The SymbolGc operations is embedded in the markForCollection algorithm and can be enabled by setting the STN_SYMBOL_GC_ENABLED = TRUE; in the stone configuration file.

Epoch and SymbolGc are covered in more detail in a later section of this document.

The function of each stage is described below.

Detection of Possible Dead Objects

EpochGc and SymbolGc are variations of the markForCollection operation. This section describes the basic operation of markForCollection, followed by additional details for EpochGc and SymbolGc.

MarkForCollection

The markForCollection operation has three phases:

a) Mark/Sweep
b) Determine possibleDead
c) Record Possible Dead

Mark/Sweep

The goal of this phase is find all of the live objects in the repository. Starting at a well-known set of root objects, these live objects are "marked". Then newly marked objects are "swept", which involves looking up the object in the object table, locating it in the cache and sweeping it for references to other objects which are also marked. This marking/sweeping process is repeated until no new marked objects are found. This traversal of the live objects in the repository is also referred to as the transitive closure.

The Mark/Sweep phase takes most of the time in markForCollection. It is important to note that markForCollection only reads object table pages into the shared page cache, not data pages. Data pages are read into a private buffer allocated by the MFC gem. This is done to avoid causing performance problems by flooding the shared cache with data pages that are of no use to other gems. More information about the page buffer is included in the tuning section of this document.

The mark/sweep phase performs a large number of disk reads. For each object swept, the MFC gem must:

- Look up the OOP in the object table to get the object's page ID. This may require the gem to read an OT page into the shared cache.
- Read the data page containing the object into the gem's mark/sweep buffer.

Executing markForCollection may cause heavy I/O and/or CPU loads on some systems.

The following factors affect the duration of this phase:

- Size of the mark/sweep page buffer.
- Number of live objects in the repository.
- Maximum disk I/O rate.
- Setting of the GEM_IO_LIMIT for the markForCollection gem.
- Size of the shared page cache.
- Size of the object table with respect to the size of the shared page cache.
- Number of other gems on the shared cache.
- The size of the commitRecordBacklogThreshold

The gem running markForCollection runs in a transaction, but aborts whenever the number of commits in the system since the last abort during the markForCollection is > 4/5 * commitRecordBacklogThreshold. Having the commitRecordBacklogThreshold set too low when there is a lot of commit activity can slow down the mark/sweep phase since the gem must abort more frequently and possibly do more object table lookups to re-synchronize with the current view of the database.

WsUnionSweep

While the Mark/Sweep phase of the markForCollection is running, the gem collects the write set union, i.e. a collection of all of the objects that were committed by other sessions while that phase was running. This phase of the markForCollection computes the possibleDead set and subtracts any of the object referenced in the write set union from the possibleDead. At the beginning of this phase it signals stone so that stone will start collecting a write set union over the commits that occur from this point until all of the gems are done with the voting phase. By having the gem doing the markForCollection collect and perform the writeSetUnionSweep at this point it saves the cost of the stone doing that work during the markSweep phase and makes the later write set union sweep performed by the adminGem much smaller.

In phase 1, the mark/sweep algorithm identified every live object. The set of possible dead objects is now computed by subtracting this set from the "universe" of object identifiers. The universe of object identifiers is defined every possible object identifier from zero up to the greatest object identifier allocated by the system (known as the "OOP high water mark"). Each object identifier in the universe can have 3 possible states:

- It can refer to a live object
- It can refer to a possibly dead object
- It can refer to no object (i.e., it is a "free OOP")

In the third case, the identifier is on the list of free identifiers and may be assigned to an object at some future time.

A "differencing" operation subtracts all live object identifiers (including the free ones) from the universe of objects, giving a resultant set that contains object identifiers that refer to possible dead objects. This operation is very fast, operating entirely on data structures in RAM.

The WsUnionSweep phase starts with the writeSetUnion collected during the first phase and considers them marked. The algorithm for this phase is similar the the Mark/Sweep, the major difference is that if a marked object is in the possible dead set, it is removed and then added to the set of objects to be swept. Again this marking/sweeping process continues until no new marked objects are found.

Since this is limited mark/sweep operation, the factors that affect its performance are similar to the ones listed for the mark/sweep phase, however the primary factor is the size of the writeSet union. Since this union is usually a small fraction of the size of the entire repository this phase is usually quite fast.

Record Possible Dead

This is the last phase of the markForCollection or epochGc operation. The gem sends a message to the stone process with a list of all possibly dead objects. The stone writes this set to the root page of the repository and then performs a checkpoint to insure that the work of detecting the possible dead objects is not lost even if the stone crashes. This phase is very fast and shouldn't significantly affect system performance (if the possible dead set is very large it can somewhat impact processing of other messages to stone and the checkpoint at the end of the operation can cause an increase in disk I/O).

Note that at this point no space has been reclaimed, no objects have been removed from the database and no objectIds have been recycled.

EpochGc

EpochGc is another technique used in the Gemstone system to determine possible dead objects. The goal of an epoch is to detect objects that have a short lifespan, i.e., that die young. An epoch is a period of time over which transactions are analyzed to determine which objects have been both created and deferenced within the epoch. The epoch is quite efficient at detecting this kind of garbage objects because it doesn't need to read the entire database to determine whether a newly created object is possibly dead. However, it is not a substitute for periodically running the markForCollection since it cannot detect objects that are created in one epoch and are dereferenced in another. The time for an epoch should be adjusted to correspond to cover a range in which many short lived objects are created to be most effective.

There are two adminGem configuration options which control the length of an epoch:

1. epochGcTimeLimit: Controls the maximum frequency of epochs, in seconds. It is recommended that this value not be less than 1800 (i.e. 30 minutes). Ideally epochGcTimeLimit should be several hours. Default is 3600 (1 hour).

2. epochGcTransLimit: Minimum number of commits that must have occurred in order to start an epoch garbage collection. Default value is 5000.

When the EcpochGc is enabled with either STN_EPOCH_GC_ENABLED or the runtime StnEpochGcEnabled configuration option, then the stone automatically collects the epochWriteSetUnion and epcochNewObjsUnion from the commits that occur. When either the timelimit or the transLimit are reached the AdminGem requests the epochWriteSetUnion and epochNewObjsUnion from stone. Stone atomically saves that state and starts a new set of unions for the next epoch.

The algorithm for the epochGc is very similar to the WriteSetUnionSweep, the major difference is that the possibleDead for the epochGc is the newObjsUnion and the starting set of marked objects is the epochWriteSetUnion minus the newObjsUnion. The mark/sweep reads the marked objects and if an object in the newObjsUnion is referenced it is removed. When the mark/sweep is done, the objects that remain in the newObjsUnion are the young

objects that died young, i.e., they were created during the epoch and by the end of the epoch any reference to them had been overwritten. This remaining newObjsUnion is then considered as the possibleDead set and is sent to stone with the same Record Possible Dead that was described above.

SymbolGc

As mentioned earlier in this document, the detection of possible dead symbols is embedded in the markForCollection algorithm and can be enabled by setting STN_SYMBOL_GC_ENABLED = TRUE; in the stone configuration file. Unless your system creates a large number of garbage symbols, SymbolGc is only needed infrequently. Enabling SymbolGc adds a couple of additional steps to the markForCollection method and to the Voting process.

When markForCollection and symbolGc is enabled, the first step is to do a mark/sweep of just the AllSymbols dictionary. This is done to collect all possible symbols into a set, allSymbols. In preparation for the normal Mark/Sweep step, the marked and swept sets are cleared of all of the symbols, leaving the existing internal objects for the AllSymbols dictionary in these sets. Then the normal roots for the Mark/Sweep are added to the marked set and the Mark/Sweep is executed.

When the normal Mark/Sweep and wsUnion step have completed, the marked set contains only the live symbols, i.e., only those symbols that are reachable from objects other than the AllSymbols dictionary. A difference operation is performed subtract the marked objects from allSymbols which results in the possibleDeadSymbols.

These possibleDeadSymbols are then sent to stone in the same message as the other possibleDeadObjects. This completes the detection of the possible dead symbols.

When stone receives the recordPossibleDead command which includes possibleDeadSymbols, it saves them and signals the SymbolGem that it has possibleDeadSymbols for it to process. The SymbolGem “hides” the possible dead symbols by removing them from the AllSymbols dictionary. When all of the possibleDeadSymbols have been hidden and the new state of AllSymbols has been committed then the normal voting and wsUnion processing are allowed to proceed. It should be noted that when the possibleDeadSymbols have been moved out of the AllSymbols dictionary, any gem that attempts to create a symbol with the same value will cause the SymbolGem to “resurrect” the symbol, i.e., take the symbol out of the hiddenSymbols dictionary, return it to AllSymbols and return the original OOP for that symbol.

When the voting and AdminGem wsUnionSweep have completed, the atomic promote step sends the symbols that have been voted to be kept alive to the SymbolGem, where they are unhidden, moved back into the AllSymbols dictionary.

Any symbols remaining in the possibleDeadSymbols, i.e., the ones that were not voted to be kept alive are now dead Symbols, which can be treated like any other dead object for reclamation by the ReclaimGem.

Voting

Every gem logged into the database must "vote" on all possibly dead objects. The purpose of the vote is to determine if the gem holds references to any possibly dead objects in any of its caches. If it does, those objects will be "voted down" and will be removed from the possible dead set by the stone. Each gem casts its vote at the next commit or abort after the *record possible dead* phase completes. All sessions must vote before this stage is completed and the next phase can be started. Therefore a gem running a long transaction (that is, one which commits or aborts infrequently) will delay completion of the voting stage.

Gems on the same machine as the stone use a shared copy of the possible dead set in the shared page cache. Gems on remote machines use a shared copy in the remote cache, however the first gem to vote incurs the cost of reading the possible dead set across the network into the cache.

Several factors determine the duration the voting phase:

- Average transaction length.
- Size of the possible dead set.
- Number of gems logged in during the vote.
- Network bandwidth between the machine running the stone process and the machine(s) running the remote gem(s), if any.

AdminGem Write Set Union Sweep

This stage starts when the last gem completes its vote by doing a commit or abort. During this phase, the AdminGem votes on objects that were committed since the mark/sweep phase of the markForCollection finished. If there were no commits during the gem voting interval then the write set union is empty and this stage is skipped.

This phase is like the wsUnionSweep that is performed by the gem that does the markForCollection, only for this operation the input set of marked objects is the writeSetUnion from stone. This operation is performed in a transaction. The adminGem does not execute an EpochGc operation while the WsUnionSweep is in progress.

This operation is generally fairly fast, but the execution time is dependent upon:

- The number of objects modified in committed transactions during voting.
- Number of possibly dead objects.
- Maximum disk I/O rate.
- Size of the shared page cache.
- Setting of the GEM_IO_LIMIT for the AdminGem.

This is the last finalization phase for the possible dead objects. Objects remaining in the possible dead set at the completion of this phase are considered to be dead objects.

Promote to Dead

The promote to dead phase "promotes" each dead object to a dead state. This is where the possible dead objects become dead objects. The promotion to dead is atomic; that is, all objects are promoted to dead by the stone in a single operation. The promote to dead stage is performed very quickly and does not require optimization.

Reclamation

The reclamation process involves recycling of the space used for objects that have been garbage collected or shadowed, i.e., replaced with an updated copy of the object. The dead objects that are reclaimed also have their Oop recycled.

During normal operation of the system, every commit shadows one or more objects and a new version of the object is written to a new page. When a transaction commits, the shadow object is no longer referenced from the new object table, and its space can be reclaimed. To facilitate knowing which pages have shadowed objects in them, when a new version of an object is written during a commit the page in which the old version of the object was located is noted and recorded in the commit record as a scavengable page. If the page contains more than 3KB then the page is marked as a priority page for reclaiming. This is to optimize the reclaiming of pages that have more free space. When the commit is processed in stone, the scavengable pages combined into a bitmaps in the root page and subsequently passed to the reclaim gem along with the priority pages for processing.

Dead objects detected by either markForCollection or epochgc are also reclaimed by the ReclaimGem but they must be looked up in the object table to determine which page they are in. For this reason dead object reclamation is considered a lower priority and the pages containing dead objects are only looked up when the pages need reclaiming set that it gets from stone is empty. When a page is reclaimed, all the "live" objects on the page are copied to a new page and any dead objects found are kept in a bitmap of reclaimedOops which is recorded in the commit record for the reclaim. To prevent page fragmentation, the new page is then "topped-up" with other live objects from other pages being reclaimed, but still honoring any clustering information in the page. The original page now contains only dead or shadow objects and may be reclaimed. Pages are reclaimed in batches, the sizes of which are governed by the "reclaimMaxPages" and "reclaimMinPages" settings in the ReclaimGem. The ReclaimGem runs inside a transaction when performing reclaims and commits after each batch of pages. The physical space contained in a batch of reclaimed pages is recovered only when the batch has been committed and the commit record for that batch has been disposed by the stone.

Page reclamation is an expensive process that may affect system performance. There are a few reasons for this. The ReclaimGem requires frequent access to a large portion of the object table to lookup the objects it is reclaiming. Consequently, the percentage of the object table that is contained in the shared cache will directly affect the performance of this operation. The amount of object table that is cached is a function of the cache size, size of the

object table, and what pages other gems are needing to access. Since live objects are copied onto a new page during reclamation, the ReclaimGem must allocate new pages during the reclaim process and these are allocated in the shared cache. Since they are dirty pages they tend to live in the cache until the aio page servers can write them to disk and increase the contention for free cache slots.

Shadow objects (generated by gems commiting modifications to objects) have a higher priority for reclamation than dead objects. When the ReclaimGem is reclaiming pages, the maximum number of pages it processes per transaction is defined by the reclaimMaxPages setting. Dead objects are reclaimed only if the number of pages needing reclaiming is less than this parameter, or if they happen to reside on the same page as a shadowed object. In the latter case, shadowed and dead objects have the same reclaim priority and both will be reclaimed at the same time. For example, let's suppose the reclaimMaxPages parameter is set to its default value of 200 pages and there are 150 pages that contain shadowed objects. If there are pages with dead objects to reclaim, the ReclaimGem will reclaim 50 pages containing dead objects bringing the total pages reclaimed to 200 for this transaction. Note that if there had been 200 or more pages available for reclaim which contained shadowed objects, no dead objects would be reclaimed during this transaction. A GcUser Boolean parameter called *reclaimDeadEnabled* allows the reclaim of dead objects to be disabled entirely.

Dead object reclaim performance is affected by several factors:

- reclaimMaxPages value
- Setting of the GEM_IO_LIMIT for the ReclaimGem.
- Size of the shared page cache
- Percentage of the object table contained in the shared cache.
- Number of shadow objects generated by other sessions' commit records.
- Number of free frames remaining on the free frame list.
- Size of the commit record backlog.
- GEM_FREE_FRAME_LIMIT parameter used by the ReclaimGem.

Returning OOPs and Pages to Free Pools

When the commit record from the page reclaim operation is disposed by the stone, the reclaimedOops are put back into the free pool and are available for use for new objects.

Since reclaimed pages were in most cases allocated in an earlier checkpoint, they cannot be returned to the free pool when the commit record is disposed. The only way pages can be recycled in the system is if they are no longer referenced.

The main inhibitor to returning OOPs and pages is a commit record backlog. If commit records are not being disposed in a timely manner, no free space will be gained from the garbage collection.

Disposing Commit Records

Commit records are linked in a chain from the current to the oldest referenced. When the last session referencing the oldest commit record aborts or commits, that commit record can be disposed. In addition to the commit record, the pages that were shadowed by that commit are also eligible for disposal. The pages that were shadowed by the commit are stored in two bitmaps referenced from the commit record, the shadowedCheckpointPages and the shadowedImmediatePages. The shadowed immediate pages can be put in the set of pages to be disposed immediately. The shadowedCheckpointPages are put into another set, whose disposal is deferred until after the next checkpoint is completed.

The reason these must be deferred is that they may be required to recover the database if the system crashed before the next checkpoint is written. When the checkpoint is written, the pages then become immediately disposable.

Before the stone can recycle the pages, it must make sure that there are no instances of the page in any of the shared page caches associated with the system. This is required to insure that a gem is not able to access an old version of the page. Stone does this by sending a list of pages to remove from the cache to the page manager. The page manager looks up the page in the cache and if it is present it tries to remove it from the cache. It may not be possible to remove immediately because it might be locked waiting for I/O to complete. The page manager then returns a list of the pages that it knows are not in the cache and stone then updates the allocated pages structures (the fragment pages) and adds them to the pool of free pages that it manages in memory.

Garbage Collection Monitoring

The progress of the various garbage collection stages may be monitored by collecting and analyzing database statistics. These statistics are useful in determining the phase of garbage collection currently running as well as the rate at which it is progressing.

VSD is the graphical tool that displays statistics that are collected using statmonitor. VSD is distributed with each version of GemStone. VSD is also available for download separately, which is useful if you wish to view statmonitor on a different platform, such as Windows.

VSD can load statmonitor data files, and if configured, allows live-monitor a running GemStone system, by starting statmonitor and automatically updating the display as statistics are collected. See the VSD User's Guide for many options for setting up monitoring.

Monitoring the Progress of Garbage Collection

There are a number of statistics that are generally useful in monitoring the performance of the system. These are listed first, then relevant statistics for each garbage collection phase are listed. The process for which the statistic is collected for is shown in brackets.

General Statistics

- **FreeFrameCount (ShrPcMonitor)**
  The number of unused page frames in the shared page cache. Gives an indication of cache utilization.

- **CommitRecordCount** (Stone)
  Although page reclaim is the final phase of garbage collection, the space occupied by reclaimed pages is not returned to the repository until the commit record from the reclaim is disposed of. Commit record backlogs will therefore delay the return of free space, which is the ultimate goal of garbage collection.

- **FreePages** (Stone)
  The number of free pages that exist in the repository.

MarkForCollection (markForCollection gem)

Mark/Sweep

- **Progress Count** (markForCollection gem)
  During this phase, the progress count displays the number of live objects that have been marked by the mark/sweep algorithm. The final value of progress count is a measure of the number of live objects found by the mark/sweep algorithm.

WsUnionSweep

- **Progress Count** (markForCollection gem)
  At the beginning of this phase, the progress count is reset to 0 and the count reflects the number of live objects removed from the possible dead set.

Record Possible Dead

- **PossibleDeadSize** (Stone)
  This statistic indicates the approximate number possible dead objects.

Voting

- **VoteNotDead** (each gem)
  Shows the number of objects voted down by the gem.

AdminGem Write Set Union Sweep

- **GcPossibleDeadSize** (Stone)
  This is an exact measure of the number of possible dead objects that survived the voting phase.

- **GcSweepCount** (Stone)
  Number of times the AdminGem has performed the write set union sweep since it started.

- **GcPossibleDeadWsUnionSize** (Stone)
  The number of objects that must be swept by the AdminGem.

- **ProgressCount** (GcGem)
  Number of objects swept thus far, can be larger than the GcPossibleDeadWsUnionSize because it includes objects found in the transitive closure that were in the possible dead.

Promote to Dead

- **PossibleDeadObjs** (Stone)
  These statistics will fall to zero at the completion of this phase.

- **DeadNotReclaimedObjs** (Stone)
  Shows the number of objects that have been promoted to dead but have not yet had their pages reclaimed by the ReclaimGem.

Reclamation

- **PagesNeedReclaimSize** (Stone)
  The number of pages needing reclaiming.

- **DeadNotReclaimedObjs** (Stone)
  The number of objects that have been determined to be dead but have not yet been reclaimed. This value will decrease as pages containing dead objects are reclaimed.

- **ReclaimCount** (Stone)
  The number of reclaims performed by the ReclaimGem since the Stone was last started.

- **ReclaimedPagesCount** (Stone)
  The number of pages reclaimed by the ReclaimGem since the Stone was last started. The count indicates the number of pages that have been or will soon be put back in the page free pool.

- **ReclaimedSymbols** (Stone)
  The number of symbols that have been reclaimed since the stone was started.

Tuning Garbage Collection

Along with the scaling changes that went into Gemstone64 that increased the size of the repository in both physical space and number of objects were changes in the scalability of the garbage collection algorithms. One of the key changes to garbage collection was to use multiple threads to increase the amount of CPU and I/O that could be brought to bear on the problem, with the goal of being able to use more of the machine resources that would be available at times when the load from user activity was less.

Although there are other factors that impact the performance of the garbage collection algorithms, the most significant one is the number of threads used to perform the operation. In addition to allocating the threads, there are also other memory resources associated with the thread that need to be allocated. To insure that the operation can be performed, the resources are allocated at the beginning of the operation with a specified maximum number of threads. If during the operation the system the system resources are needed for another task, the number of active threads can be decreased, and then later increased back to the original maximum number of threads. The number of threads currently running an operation in a gem session can be accessed using the method Repository >> mtMaxThreads: sessionId. The value can be changed using Repository >> mtThreadsLimit: sessionId setValue: newVal.

Changing the number of active threads generally impacts both the CPU utilization and the I/O rate for the gem process. Another alternative to managing the CPU utilization is to change the setting for the mtPercentCpuActiveLimit using the method Repository >> mtPercentCpuActiveLimit: sessionId setValue: newVal. The multi-threaded operations periodically monitor their CPU usage and and automatically adjust the mtThreadsLimit to keep the utilization below the specified value.

An examination of the system should be performed before any tuning is performed. The goal of such an examination is to determine the cause and nature of system latency. The following questions should be answered to aid this analysis:

- What aspect of system performance is being affected? Examples include CPU availability, I/O bandwidth, disk swapping, and network I/O bandwidth. UNIX tools such as top, vmstat, iostat, netstat, sar and glance may be used to assist in this determination.
- What aspects of GemStone performance are being affected? Examples include commit performance, system login time, etc.

Once system resource limitations are understood and the tuning goals determined, the various areas of garbage collection can then be addressed to achieve those goals.

The next sections of this document covers tuning details specific to individual operations.

markForCollection

The method Repository >> markForCollection performs the operation in a non-aggressive manner and uses only 2 threads. The Repository >> fastMarkForCollection method performs the operation more aggressively in order to complete the operation in as little time as possible. The default maxThreads for this method is the number of CPU cores in the system * 2. It is expected that many of the threads would be in I/O wait when scanning large repositories and will likely consume most or all of the host system resources while it is in progress. The number of threads used can also be specified explicitly using the method Repository >> markForCollectionWithMaxThreads: numThreads.

There are a number of memory overheads associated with running the markForCollection method. The memory space needed is variable and depends on the current oopHighWaterMark, the number of threads and the pageBufSize specified. The markForCollection algorithm doesn't use any TemporaryObjectCache space, so configuring this gem for the smallest TOC space is advantageous.

The estimated overhead associated with the oopHighWaterMark can be computed as:

BitArrayBuffersSizeInBytes = (stnOopHighWater + 10M) / 2

The estimated memory cost per thread is 50K + (180K * pageBufSize). The default pageBufSize is 128 pages.

To give some idea of how this scales, a system that had an oopHighWaterMark of 500M running 8 sessions with a pageBufSize of 512 would require about 1GBof free memory to start up.

Caution should be taken when setting the numSessions and pageBufSize to prevent the memory footprint for this process from being so large that the system starts swapping.

The pageBuffer is used to hold data pages read by the thread. The thread loads the page buffer with pages it has identified as containing marked objects looking up the objects in the object table to make sure it is viewing the current state of the object, this phase is very I/O intensive. It then performs a mark/sweep of the objects in those pages which is CPU bound since no additional I/O is needed. In general, a larger buffer allows for the possibility of finding new marked objects already in the buffer, but there are no guarantees that this will occur, since it is largely dependent upon how the objects were clustered and what pages the thread happens to get in its buffer. The pageBufSize specified must be a power of 2. Larger values can improve performance. For example, in a test case where the machine was dedicated to performing this operation, increasing the pageBufSize from 64 to 512 reduced the elapsed time by about 20 percent. On the other hand, if the operation is being run on a live system then smaller values may be better depending on how often the algorithm must abort (and clear the buffers) to prevent the commit record backlog from growing. The default pageBufSize is 128 pages.

During the mark/sweep the repository is scanned repeatedly from low pageId to high pageId within each extent and each thread attempts to select a set of pages from a different extent. If the extents are on disk drives, performance can be optimized by arranging the extents so that the scan causes the fewest head movements on each drive since the threads can simultaneously initiate I/O operations on multiple extents.

For use with a RAID or SAN system, the device should be configured for 16K read operations from sequentially increasing blocks within an extent. No tuning should be necessary with SSDs.

The markForCollection requires almost continuous access to the entire object table. Running on a system that can hold all of the object table in the shared page cache will allow it to perform optimally. The object table size is approximately stnOopHighWaterMark * 12 bytes.

Epoch GC

Tuning the epochGc can be done by changing the settings of the adminGemConfigs using the System >> setAdminConfig:toValue: method. The configs that control the epochGc are:

#epochGcMaxThreads
MaxThreads used for next epochGc
(default:1, min:1, max: 32)

#epochGcPageBufferSize
Size in pages of buffer used for epoch GC (must be power of 2)
(default: 64, min:8, max 1024)

#epochGcPercentCpuActiveLimit
Limit active epoch threads when system percentCpuActive is above this limit.
(default: 90, min: 0, max: 100)

#epochGcTimeLimit
Controls the maximum frequency of epochs, in seconds. It is recommended that this value not be less than 1800 (i.e. 30 minutes), since the aging of objects faulted into gem memory uses 5 minute aging for each of 10 subspaces of the pom generation. Ideally epochGcTimeLimit should be several hours.
(default: 3600, min: 5, max: maxInt32)

#epochGcTransLimit
Minimum number of commits that must have occurred in order to start an epoch garbage collection.
(default: 5000, min: 0, max: maxInt32)

The epochGcTimeLimit and epochGcTransLimit determine the span of the epoch and thus the size of the writeSetUnion that must be processed. To have the epochGc be effective, these should be set so that the duration of the epoch is at least 2 times the lifetime of the objects you would like to detect.

The execution time for the epoch processing is most effectively controlled with the epochGcMaxThreads configuration parameter. Having more threads operating will shorten the processing time required, but can increase the CPU and I/O load on the system. If the epoch can be scheduled to run in off load times, for example: during a change of shifts or overnight, then a higher value can be used.

If the epoch needs to be run during the normal business operations, then increasing the number of threads may not be possible, but the performance of the fewer threads may be improved by increasing the epochGcPageBufferSize.

If an epochGc operation is consuming too much CPU or I/O it can be dynamically tuned by reducing the epochGcPercentCpuActiveLimit.

Symbol GC

There are no specific changes that can be made to tune Symbol garbage collection. The additional steps that occur during the markForCollecion are controlled by the configuration for the markForCollection. These phases in the markForCollection are usually much smaller than the full mark/sweep phase that no additional tuning is required.

The hiding of the possibleDeadSymbols by the symbol gem is probably the most critical factor, but this is currently a single threaded operation that cannot be tuned.

The best advice is to run the symbol garbage collection in a maintenance window when there are fewer active gem processes on the system.

Voting

Ordinarily no tuning is required for this phase. Voting will complete faster if gems are running in short transactions and the STN_CR_BACKLOG_THRESHOLD configuration parameter is set to a smaller value to initiate sending of SIG_ABORT to idle gems.

A "brute force" method for completing this phase quickly is to stop and restart the stone.

WriteSetUnonSweep

Like epochGc, the writeSetUnionSweep can be tuned by changing the settings of the adminGemConfigs with the System >> setAdminConfig:toValue: method. The configs that control the writeSetUnionSweep are:

#sweepWsUnionMaxThreads
MaxThreads used for next wsUnion sweep
(default: 1 min: 1, max: 32)

#sweepWsUnionPageBufferSize
Size in pages of buffer used for wsUnion sweep.
(default: 64, min: 8, max: 1024)

#sweepWsUnionPercentCpuActiveLimit

Limit active wsUnion threads when system percentCpuActive is above this limit.
(default: 90, min: 0, max: 100)

The execution time for the wsUnionSweep processing is most effectively controlled with the sweepWsUnionMaxThreads configuration parameter. Having more threads operating will shorten the processing time required, but can increase the CPU and I/O load on the system.

Since it is difficult to time when the wsUnionSweep will be run, increasing the number of threads may not be desirable, but the performance of fewer threads may be improved by increasing the sweepWsUnionPageBufferSize.

If the wsUnionSweep operation is consuming too much CPU or I/O it can be dynamically tuned by reducing the sweepWsUnionPercentCpuActiveLimit.

PromoteToDead

The promote to dead step is an atomic operation that is executed by the stone process. It involves copying the possibleDead bitmap to the deadObjects bitmap, which generally is a very fast operation so no tuning is required.

If symbolGc is enabled, the stone also wakes up the symbol gem to unhide the symbols that are not dead.

Reclamation

The reclaim processing can be tuned by changing the settings of the reclaimGemConfigs with the System >> setReclaimConfig:toValue: method. The configs that control the reclaim process are:

#deadObjsReclaimedCommitThreshold

The maximum number of dead objects to reclaim in a single transaction. Increasing this value can make the commits more efficient, but may cause the reclaim gem to hold the commit token for a longer time than desired.
(default:20000, min:32, max: maxInt32)

#deferReclaimCacheDirtyThreshold

Specifies a threshold percentage of dirty pages in the stones shared page cache. If more than this percentage of pages in the stone shared cache are dirty pages, page reclaims will be deferred until the dirty percentage drops below the threshold minus 5%. Setting the value to 100 disables this feature.
(default:75, min:10, max: 100)

#maxTransactionDuration

Controls the approximate maximum length of a reclaim transaction in seconds. If the Reclaim gem has been in transaction longer than this duration, it will commit even if the #objsMovedPerCommitThreshold condition has not been satisfied.
(default:300, min:1, max: maxInt32)

#objsMovedPerCommitThreshold

Controls the approximate number of object table updates to be performed per transaction. Reclaim gems will commit when at least this many live objects have been moved to new data pages.
(default:20000, min:100, max: maxInt32)

#reclaimDeadEnabled

Controls whether the reclaim will try to clean up dead objects.
(default:true, min:false, max:true)

#reclaimMinFreeSpaceMb

Minimum repository free space, expressed in megabytes, which must be available in order for reclaims to proceed. Reclaims are temporarily suspended if the repository free space drops below this threshold. The default value of 0 specifies a varying reclaimMinFreeSpaceMb that is computed as the current size of the repository divided by 1000, with a minimum value of 5 mega bytes. The default calculation is the same as that used by the stone for the STN_FREE_SPACE_THRESHOLD configuration parameter.
(default:0, min:0, max: 65536)

#reclaimMinPages

The minimum number of pages to process in a single reclaim commit. If there are not this many pages to reclaim, the reclaim gem sleeps until there are at least this number available.
(default:40, min:1, max: maxInt32)

The above tuning parameters are useful for tuning the reclaim so that it doesn't interfere with the normal operation of the system. If there are times when the system is not being heavily used and there are a lot of pagesNeedReclaiming or deadNotReclaimed, the SystemRepository >> reclaimAll method can be used to automatically override some of the settings to optimize the reclamation. When the reclaimAll completes the settings are restored to their previous state.

Performance Examples

These examples were run with a 3.7.0 alpha version as of May 2022, using a a 48 CPU machine with Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz, running Ubuntu 18.04. The system was configured with 100 GB shared page cache and running with 2MB large memory pages.

The database was generated using large backup test. It was spread across 5 extents. Extents 0 and 1 are on SSDs and limited to 220GB, the remaining extents are on a raid file system. The allocation mode is set to 100, 100, 20, 20, 20. The extents are pregrown at startup to 220GB, 220GB, 200GB, 200GB, 200GB.

The ReclaimGem was configured to run with 12 threads.

The first operation performed was to grow the database to contain 8 billion objects. This was done using 8 gem processes and took 5 min 43 seconds to complete. The stone averaged around 250 commits per second during this time and consumed about 445.6 GB of the allocated repository space.

Then a backup, compressed using lz4 compression was run using 10 threads and written to 8 backup files on the raid file system. This took 8 minutes to run, writing 8.559 billion objects, a rate of 17.76 million objects per second. PercentCpuWait averaged around 10% for most of the backup operation while the PercentCpuActive was about 25%.

Approximately half of the objects were then disconnected and markForCollection was run using 64 threads. The mfc took 6 minutes, 38 seconds to compete and added 4.266 billion objects to the possible dead.

Because no other gems were logged into the system the voting was pretty much instantaneous.

The reclaimGem took 11 minutes 39 seconds to reclaim all of the dead objects and was committing 1 to 2 times per second.

Finally a restore operation was performed using 10 threads. The restore took 9 minutes 52 seconds to do the restore of all 8 billion objects.

These values are just an example of the possible performance for these operations and as when comparing automobile mileage, your results will almost certainly vary.