Summary

Evacuator is the work-in-progress name for a novel generational garbage collection (GC) algorithm that uses object characteristics and current operating conditions to select optimal strategies for laying out evacuated objects in the heap while maintaining high evacuation throughput. This work was started at IBM Canada and recently resumed to explore some new techniques intended to streamline stack operation.

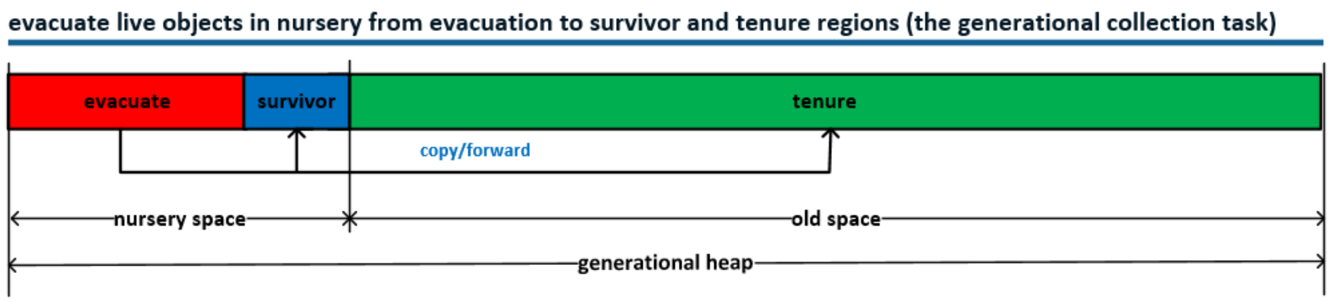
The Evacuator prototype under test works with a classical generational heap model, with a nursery comprised of an evacuation and a survivor region (which receives young evacuated objects), and a tenure region (receiving old evacuated objects). It has minimal integration points touching other JVM components and can easily be extended to subsume copy/forward work in more modern multi-generational and concurrent collectors.

OpenJ9 builds that include Evacuator can be run with Scavenger or Evacuator; either can be selected with appropriate command-line directives. Preliminary benchmarking with a subset of SPECjvm2008 benchmarks demonstrate that Evacuator outperforms Scavenger in terms of object collocation and completes fixed benchmark workloads in less time using less energy (>1% speedup vs Scavenger). Over time, and across the spectrum of servers running Java, this amounts to a significant reduction in operating cost.

This technical paper briefly describes the Evacuator prototype, presents some preliminary benchmarking results, and offers some suggestions for further work.

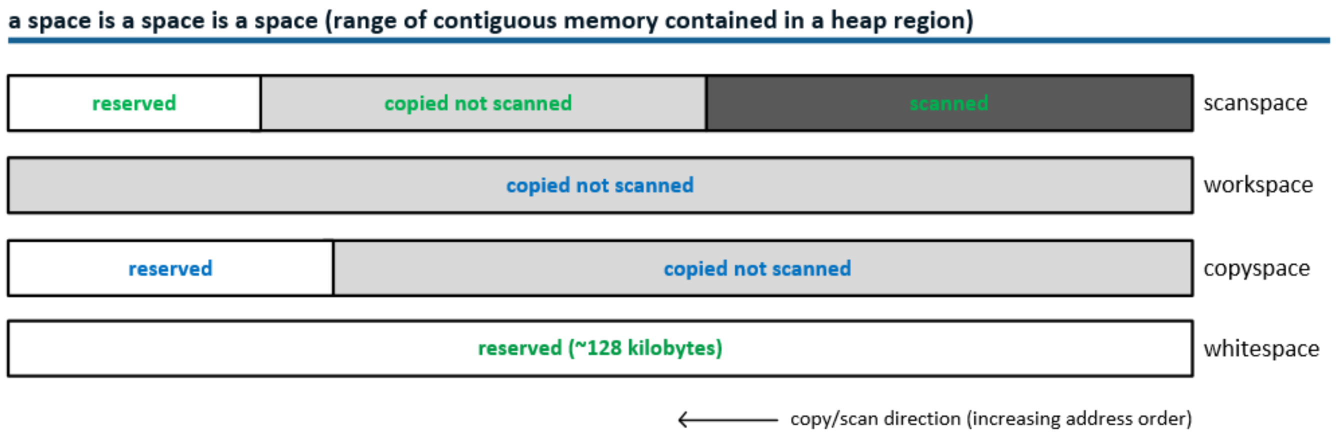
Overview

Evacuator is designed for extension to other generational frameworks and can easily be adapted to operate with >3 heap regions or regions of variable size and/or concurrently with the application as for modern collectors, such as OpenJDK's Z or OpenJ9's Concurrent Scavenger. Each region is associated with an allocator that reserves heap space for object copy and recycles unused reserved heap space. Below is a representation of the classical heap layout used for Evacuator prototyping.


evacuate live objects in nursery from evacuation to survivor and tenure regions (the generational collection task)
evacuate
survivor
tenure
copy/forward
nursery space
old space
generational heap

The central feature of Evacuator design is the scan stack, consisting of a fixed number of frames. Each frame contains an *object scanner* within a *scanspace,* a scan head and base and end points spanning a contiguous range of bytes within a survivor or tenure *copyspace*. The scan head is positioned at the head of the object being scanned, and the object scanner pulls references to evacuation space from the object for copying. Each copyspace has a base and an end point spanning a contiguous range of reserved survivor or tenure space and a copy head marking the position where the next object will be copied. The copy head partitions the copyspace into a range of unscanned copy (workspace) and heap space reserved but not yet consumed (whitespace).

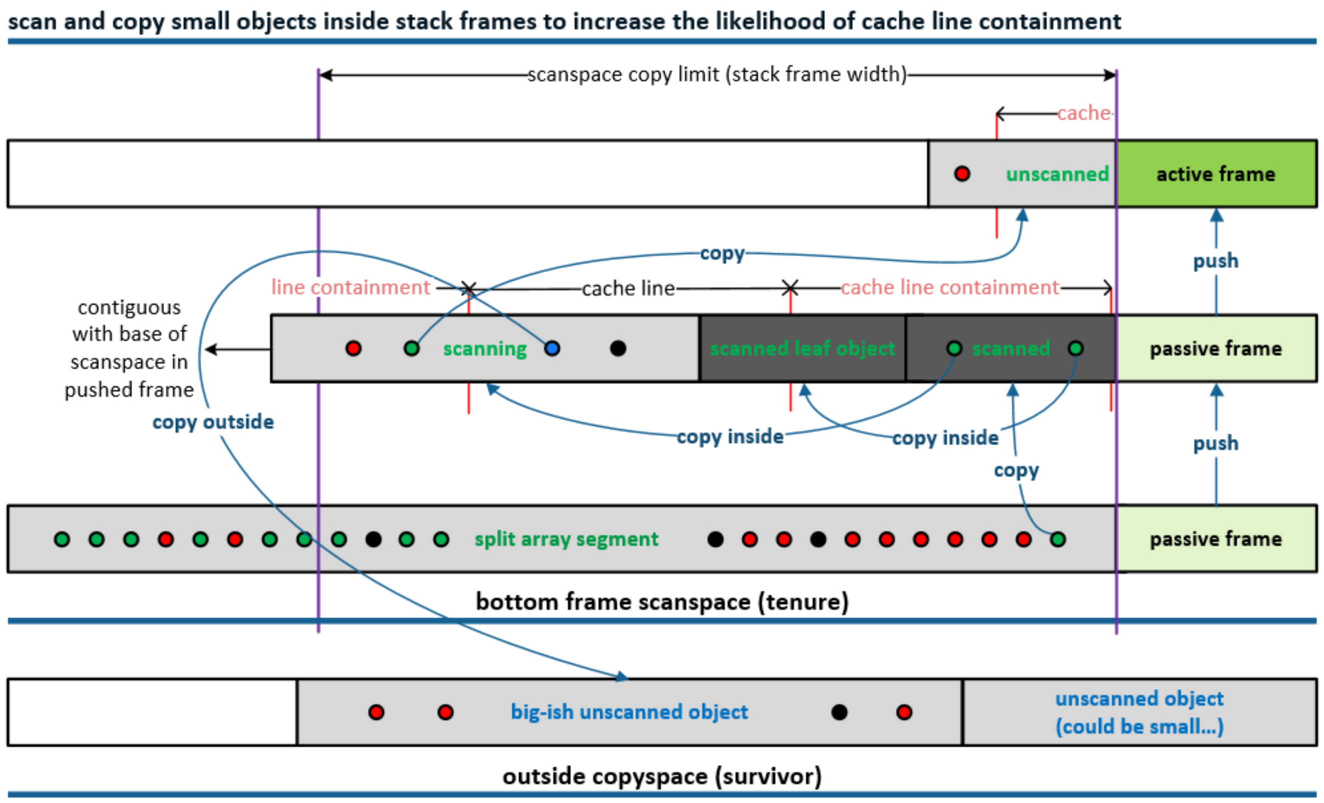
The graphic below illustrates the kinds of spaces involved in Evacuator collections (this is a bit outdated – scanspaces no longer map reserved whitespace, their endpoints are coincident with past impressions of a copyspace copy head).


a space is a space is a space (range of contiguous memory contained in a heap region)
reserved
copied not scanned
scanned
scanspace
copied not scanned
workspace
reserved
copied not scanned
copyspace
reserved (~128 kilobytes)
whitespace
copy/scan direction (increasing address order)

There are two copyspaces – *inside* and *outside* – for the survivor region and two for the tenure region. The copy head of an inside copyspace is tracked by the end point of the topmost scanscape that is scanning within the respective region. Copy proceeds until an object is copied outside the frame boundary and is pushed up the stack, freezing the scanspace end point. After scanning above and popping back into the frame scanning proceeds within the scanspace until the scan head advances to the end point and the frame is popped. The *outside* copyspaces receive objects that overflow the inside copyspace and release workspaces to a worklist that buffers deferred scan work.

The size of objects admitted for inside copying is determined by a specified maximum inside copy size (MICS). The maximum distance between the base of a scanspace and the end point is limited by a specified maximum inside copy distance (MICD). Also, leaf objects and arrays of primitive (instances of classes that contain no referent pointers) that do not require scanning are admitted for inside copying, irrespective of size, only when they can be copied with cache line containment as described below. This is a key feature, as it results in frequent recursive collocation of String objects with associated text (char[]), hash table nodes with hashed keys and values, etc.

As each evacuation reference is pulled from the object scanner the symmetric distance between the scan head and the copy head of the inside copyspace is determined by XOR'ing the respective addresses. If the result is less than the size of a L1 cache line (64 bytes for AMD64 machines), the inside copyspace is selected to receive the collocated copy. The end point of the scanspace is clipped and fixed if the distance between the base of the scan space and the head of the copied object exceeds the maximum inside copy distance (4kb by default), and the copied object is pushed into a superior frame for scanning. In that case, the end point of the scanspace in the inferior frame is coincident with the base of the scanspace in pushed frame, as shown below.


scan and copy small objects inside stack frames to increase the likelihood of cache line containment
scanspace copy limit (stack frame width)
cache
unscanned
active frame
copy
push
line containment
cache line
cache line containment
contiguous with base of scanspace in pushed frame
scanning
scanned leaf object
scanned
passive frame
copy outside
copy inside
copy inside
copy
push
split array segment
passive frame
bottom frame scanspace (tenure)
big-ish unscanned object
unscanned object (could be small...)
outside copyspace (survivor)

Objects between the base and scan head are completely scanned and are dropped from the workflow when the scan head reaches the end point and the frame is popped off the stack.

Objects not captured for inside copying are copied to the *outside* copyspace for the selected (survivor or tenure) region. When the distance between the base of the outside copyspace and the copy head passes a variable maximum workspace size (MWS) threshold the copyspace is rebased to release a workspace to the worklist and the base pointer is set to the copy head, leaving only whitespace between copy head and end point. When the volume of remaining whitespace becomes too small to be useful (<256 bytes) or overflows excessively the remaining whitespace is trimmed to a whitelist holding fragments of unused whitespace and the copyspace is refreshed with new whitespace. Each survivor region has its own whitelist, which is a priority queue retaining up to 15 whitespaces presenting largest on top. Smaller whitespaces that underflow the whitelists are marked as heap holes and recycled or discarded.

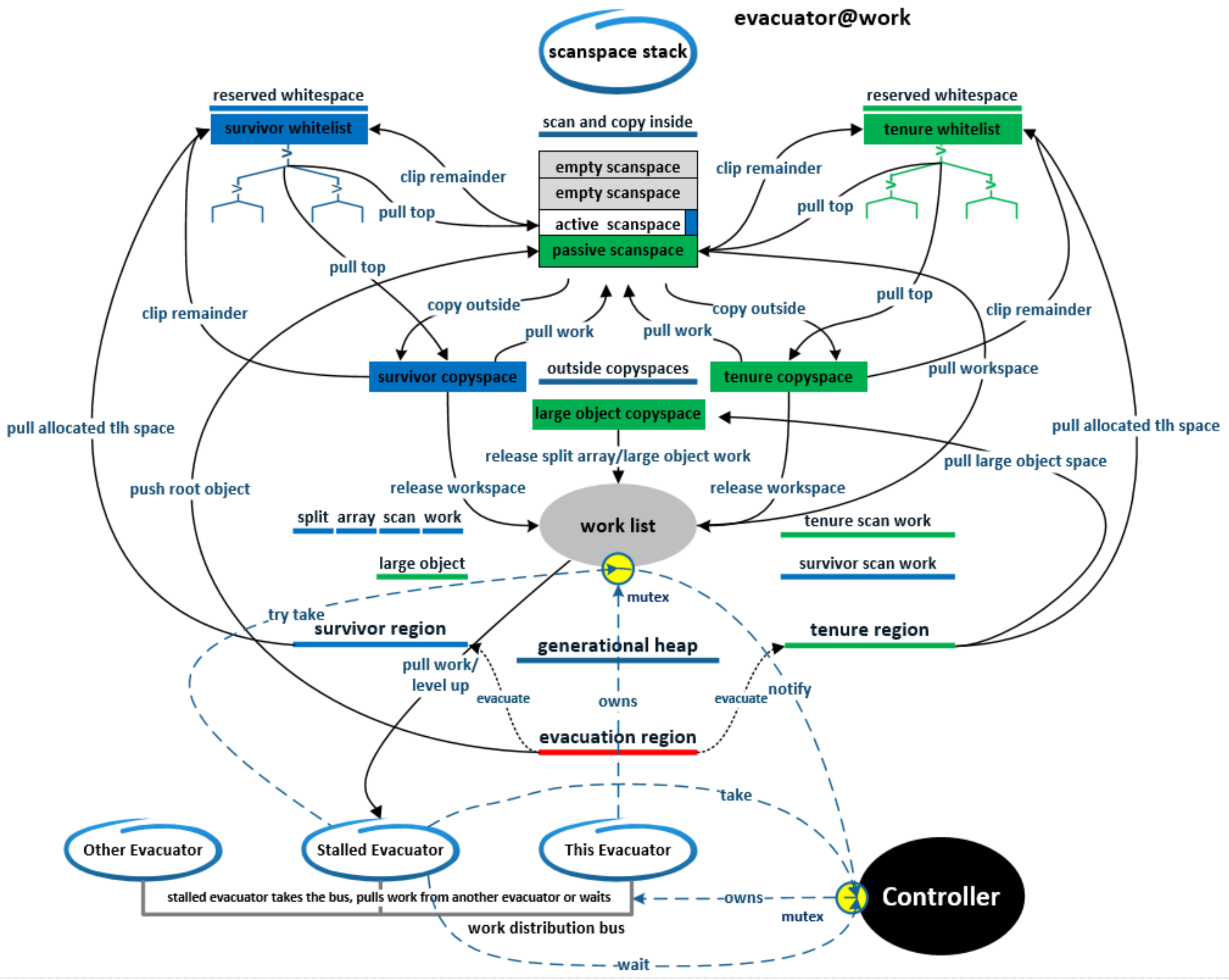
Objects that overflow inside and outside copyspaces are routed into an overflow copyspace (outside copy may be deflected back to the respective inside copyspace if it overflows outside copyspace and outside copyspace cannot be refreshed). Ordinarily, the overflow copyspace also receives leaf objects and primitive arrays that escape cache line containment and drops from the workflow after copy as they do not require scanning. But when a stall or stack overflow condition is raised these objects are copied to the outside copyspaces unless copied inside for collocation, and the MWS is reduced. This increases the rate and volume of distributable work flowing into the worklists of working evacuators, and helps to redistribute branches of recursive structures (eg XML/JSON object model) among evacuator threads when scan stacks overflow. This is demonstrated with the SPECjbb2008 xml.validation benchmark, which overflows the scan stack frequently.

Each evacuator instance has a local worklist that receives workspaces released from outside copyspaces and distributes workspaces locally and to starved evacuator threads. Each evacuator pulls workspaces from its local worklist, filling up to half of its bottom stack frames, to refresh whenever its scan stack runs dry. Only one workspace can be pulled if a stall condition is set, indicating that the evacuator’s worklist volume is below quota or that another evacuator thread has stalled and is waiting for work. In that case, the MICS and MICD are restricted to admit only very small objects for inside copying (MICD = 32 bytes, with exceptions for collocatable leaf and primitive objects of any size) and to push each inside copy up the stack (MICD = 0 bytes forces depth-first scanning). These restrictions increase the volume of outside copy and unscanned work flowing into the worklist while allowing important small objects to continue to be scanned and copied depth-first to maximize collocation.

An evacuator controller is responsible for instantiating and destroying evacuator instances. The controller also provides a work distribution bus that allows starving evacuator instances to pull workspaces directly from other evacuators’ worklists. While on the bus stalled evacuators seek a donor with a maximal volume of distributable work and pulls roughly half of the available volume. This has a general load-leveling tendency as long as there a sufficient volume of available work.

Evacuator threads must scan all work on scan stack, worklist, outside and overflow copyspaces before taking the work distribution bus to look for work. If there is no distributable work available they will stall and wait on the bus monitor for another evacuator to notify of distributable work or until all evacuators stall and wait on the bus. The last evacuator to stall releases all evacuator threads to continue to the next stage of the GC cycle.

Evacuator operation is summarized in the graphic below, which also is a bit out of date. It shows only the outside copyspaces and the overflow copyspace (labeled *survivor copyspace, tenure copyspace, large object copyspace* in the diagram), including the inside copyspaces was a late design decision. In the diagram below, the inside survivor and tenure copyspaces belong with tops vertically aligned with stack bottom, inside edges horizontally aligned with center of corresponding outside copyspace, and each with two arcs labeled *scan inside* and *copy inside* leading from (pick one) *active*|*passive scanspace.*



Also remove the *clip remainder* arcs out of *active*|*passive scanspace*, scanspaces no longer map whitespace. Workspaces are never released from inside copyspaces – all inside copy is scanned inline on the stack. All copyspaces are refreshed from whitelists when whitelist top() volume is ≥MICS but are generally refreshed with 128kb of whitespace from the region memory allocator. Otherwise allocations are occasionally made for the overflow copyspace to specifically fit large solo objects – these are released as workspaces (or dropped from workflow if primitive array) as soon as copy is complete.

Benchmarking Performance[†]

The below table presents average statistics for salient outcomes for each benchmark. The average duration of the interval between successive GC cycles (*interval-ms*), benchmark score, total run time in seconds, and CPU utilization as reported by Linux time for two runs of each benchmark are presented in the columns on the left. The columns on the right for GC performance are averages over all generational GC cycles summed over 2 runs, the *N* column shows the number of GC cycles.

derby

Xms=Xmx=768m

average benchmark performance (N=2)				gc	*average gc performance*				
interval-ms	score	time	cpu%	configuration[*]	gc-ms	kb/ms	cache%	cpu%	N
272.92	185.62	654.57	315.50	***scavenger***	3.06	296.92	13.81	345.35	4758
266.87	189.82	639.65	315.00	***MICS-4096***	3.52	251.72	36.75	242.00	4750
270.28	188.27	647.09	314.00	***MICS-1024***	3.53	251.84	36.55	244.57	4747
266.81	190.62	639.48	313.00	***MICS-256***	3.50	249.60	36.76	246.05	4752
273.14	185.75	654.03	314.00	***MICS-64***	3.55	252.35	36.67	246.30	4750

xml.validation

Xms=Xmx=1280m

average benchmark performance (N=2)				gc	*average gc performance*				
interval-ms	score	time	cpu%	configuration	gc-ms	kb/ms	cache%	cpu%	N
508.05	175.61	547.92	394.50	***scavenger***	69.64	510.09	2.46	389.41	2151
503.66	176.45	545.05	393.00	***MICS-4096***	73.24	488.92	25.17	372.98	2148
479.50	184.30	521.96	393.00	***MICS-1024***	72.38	488.55	25.20	373.46	2173
493.88	179.55	535.71	392.50	***MICS-256***	72.48	493.02	24.90	374.01	2163
486.87	183.46	524.18	392.50	***MICS-64***	74.24	486.06	24.10	375.40	2148

xml.transform

Xms=Xmx=768m

average benchmark performance (N=2)				gc	*average gc performance*				
interval-ms	score	time	cpu%	configuration	gc-ms	kb/ms	cache%	cpu%	N
358.90	131.89	750.62	389.50	***scavenger***	5.49	1,487.46	12.03	377.37	4170
349.01	135.75	729.74	389.00	***MICS-4096***	5.57	1,488.80	16.42	361.29	4170
351.67	134.72	734.98	389.50	***MICS-1024***	5.28	1,564.25	18.19	373.06	4168
352.82	134.26	737.51	390.00	***MICS-256***	5.12	1,645.18	22.68	382.61	4168
358.25	132.17	749.24	390.00	***MICS-64***	5.18	1,637.14	22.46	381.87	4171

[*]***MICS = maximum size of objects admitted for copying and scanning inside stack***

MICD-0 (maximum distance from frame base for copying inside frame, 0 forces depth-first scan)

MSD-16 (maximum depth of scan stack, 1 forces breadth-first outside copy)

MWS-4096 (minimum workspace size for deferred scan work, max is 16*MWS)

MCS-8192 (minimum allocation size for evacuator copyspace, max is 16*MCS)

† *Table is incorrect, xml.validation heap was 768m owing to an error in the benchmarking script, and there were >20 global GCs for each scavenger or evacuator run. No global GCs occurred during the xml.transform or derby runs and all other metric values were as shown.*

The *gc-ms* metric represents the difference between system clock samples taken just before and after the GC slave threads are forked and joined. The *kb/ms* metric is GC throughput (*copied-kb / gc-ms*). The *cache%* metric represents the percentage of evacuated objects collocated with 64-byte L1 cache line containment of referring pointer and head of referent object. The *cpu%* metric is average CPU utilization reported per thread from kernel *rusage()* sampled contemporaneously with the system clock for *gc-ms.*

For each benchmark the least average benchmark score for the Evacuator runs was greater than the best average score for any of the Scavenger runs, and the percentage of object copied with cache line containment was 2-3 times greater for each the Evacuator runs, versus Scavenger. Scavenger throughput is outstanding – for these benchmarks Scavenger threads rarely stalled. Working evacuator threads strive to maintain a distributable volume of work for sharing with starved evacuators, and as evacuation reaches its final stages they tend to spin on local scan work without raising worklist volume above quota. However, the execution speedup obtained by increased collocation of objects *appears* to compensate for the concomitant increase in GC time (see *interval-ms* and *time* in the table above).

The tables above do not directly show evacuation volumes *copied-kb* but they are not in agreement (*copied-kb > (N+1)*interval-ms + N*gc-ms*) with the values reported in the tables for interval and GC times. The interval-ms values are stripped from the verbose GC log and may be used inappropriately here. In future reports this will be measured as intended directly, but for present purposes these values can be viewed as comparable in a relative sense.

Fragmentation within the nursery and wherever tenure copy is laid down is negligible, with <10^{-3} percent of reserved whitespace discarded in tenure or survivor space, due to the inclusion of whitelists as described above. Inside copyspaces cannot be refreshed until <32 bytes of whitespace remain. Outside copyspaces are driven to <256 bytes unless there is a large volume of overflow. Whitespace fragments trimmed from outside and overflow copyspaces are always presented to the appropriate whitelist, which retains the 15 largest fragments and discards overflow and all fragments of <64 bytes. Large whitespace remainders in survivor copyspaces when an evacuator instance completes are recycled back into the memory pool and are available for reuse by the application. Tenure whitelists retain their contents between generational cycles and are recycled back into the only in the event of a global GC, where they are collected and recycled for application use.

Going Forward

There are number of features yet to be developed and tested in the Evacuator framework. Generally, Evacuator tries to scan evacuated objects in the same order as they are *copied*, Ideally, they should be scanned in the same order in which they are *allocated*. The OpenJ9 stack walker presents thread slots (stack frame references to heap objects) to evacuators in top-down order. Presentation from the bottom up would tend to trace the natural order in which parent-child reference arcs are formed and would produce more frequent and salient collocation of objects.

Generational collectors have boundary issues that emerge when the available whitespace in the survivor or tenure region runs low. When survivor space is exhausted and the memory allocator for the survivor region fails to deliver a TLH or whitespace for a solo object copy, the requesting thread is forced to allocate from tenure. For Scavenger there is no other option since any reserved whitespace that it might be holding is inaccessible (Scavenger scan/copy structures in deferred worklists hold reserved whitespace but it is not accessible for reuse).

Evacuator instances can access and reuse whitespace bound within their own copyspaces but this is a strictly limited resource. However, the survivor/tenure barrier is seldom breached before nearing the end of the evacuation cycle and if one or more other evacuators are stalled they should be able to yield their whitespace to keep other evacuators, possibly, from breaching the survivor/tenure barrier and overflowing young objects into tenure space. This may prevent the collector from making unwarranted changes to the heap configuration to deal with transient conditions.

Similarly, and more importantly, the memory allocator for the tenure region may also run dry, precluding any further new whitespace allocations in any region. Currently, Scavenger and Evacuator both abort in this situation, with the consequence that a time-intensive single-threaded back-out algorithm must be executed to undo the evacuation so that a global GC may run before the application can proceed. To reduce the likelihood of this event, evacuator instances that fail to allocate whitespace from the memory allocator should first strip stalled evacuators of whitespace, assume their outstanding scan work, and attempt to continue, possibly completing without back-out but signaling that a global GC is required. This would likely reduce the frequency of back-out and consequent disruption of application throughput, which is something that everyone would like to avoid.

Finally, Evacuator tracks a rich set of metrics that could be used to characterize an application's workflow and adapt GC operation to optimize collection and application throughput. Each evacuator thread is the sole operator of an evacuating automaton. As each object is presented for evacuation the operator receives some characteristics of the object (shape, size, age, leaf, hottest reference field?, …) and an array of metrics describing the state of the evacuation process. It uses this information to direct the action of the evacuating automaton and effect collocated object copy. Further refinement in this regard is certainly possible, and the metric data from trace-enabled Evacuator builds is a starting point for exploring this. To what extent can the application data direct its own collection?

Conclusion

Benchmarking with the current prototype suggests that an application speedup of >1% can be realized with the Evacuator algorithm, making it a good candidate for GC in applications that are throughput oriented. But the real motivation for continuing to investigate this algorithm is to repurpose it for use in more modern region-base generational collectors like OpenJ9's balanced collector OpenJDK Z. Adaptation for use in concurrent collectors should not be discounted – evacuator threads can be paused after completing (or kicking off) any copy-forward operation and restarted at a later time, or can be stripped of resources (unscanned work, unused whitespaces) whenever they are paused or stalled – these resources can then be reallocated to other evacuator instances or reused in other ways.

Appendix – Building and Running

Evacuator repos are here:

https://github.com/ktbriggs-gc/omr - (evacuator-redux branch)

https://github.com/ktbriggs-gc/openj9 - (evacuator-redux branch)

These branches and my openj9-openjdk-jdk8 repo are a bit out of date with the respective masters (last pulled ~June 17).

```
$ gitc openj9-openjdk-jdk8 log --oneline -n 1
```

```
61d22079a8 (HEAD -> openj9) Merge pull request #396 from keithc-ca/close_mutex.
```

```
$ /root/bootjdk8/bin/java -version
```

```
openjdk version "1.8.0_242"
OpenJDK Runtime Environment (build 1.8.0_242-b08)
Eclipse OpenJ9 VM (build openj9-0.18.1, JRE 1.8.0 Linux amd64-64-Bit Compressed
References 20200122_511 (JIT enabled, AOT enabled)
OpenJ9 - 51a5857d2
OMR - 7a1b0239a
JCL - 8cf8a30581 based on jdk8u242-b08)
```

I always configure with **–disable-ddr**. Other than that, standard configure & make should do it.

A caveat to anyone who tries this: Testing to date has been limited to hours of grinding on three SPECjvm2008 benchmarks with various heap configurations, including very small fixed heap size (forcing multiple aborts), default variable heap sizes (2MB and rising), and large-ish (~1G) heaps.

In Evacuator builds of OpenJ9 Java, scavenger will run for gencon by default. To run OpenJ9 Java with evacuator enabled, select **-Xgc:recursiveScanOrdering**. Other evacuator options are:

-XXgc:recursiveMaximumStackDepth (>0, 1=breadth-first copy always[1], 16=default)

-XXgc:recursiveMaximumInsideCopySize (>16, ≤4096, 4096=default)

-XXgc:recursiveMaximumInsideCopyDist (0=depth-first scan, ≤65536, 4096=default)

-XXgc:recursiveScanOptions (0=default, 16=breadth-first copy all root/remembered objects)

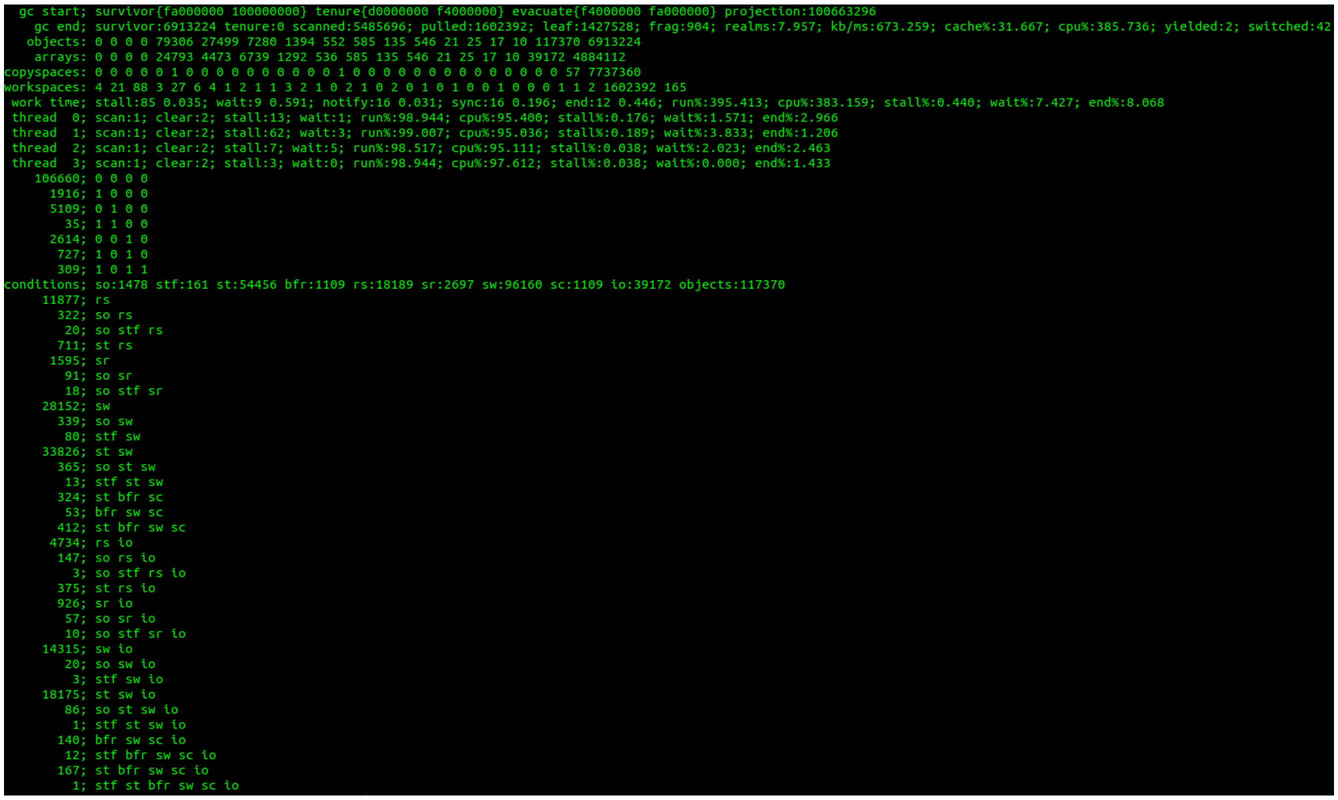
-XXgc:recursiveTraceOptions (0=default, 1 prints per gc metrics if tracing enabled in the build)

Scan and trace options are additive. Additional trace options include 128 (whitespace allocation) and 512 (fill allocated whitespace with holes and verify all copy/forward reservations and all fragments trimmed from exhausted copyspaces are unused whitespace). Tracing is enabled in evacuator builds only if **EVACUATOR_DEBUG_ALWAYS** or **EVACUATOR_DEBUG** is defined in EvacuatorBase.hpp. These are (should) not be defined in committed source, so must be edited in if desired.

For anyone planning to make changes, I recommend defining **EVACUATOR_DEBUG** in EvacuatorBase.hpp and compiling a debug build (there is a blog post describing how to do this here: https://blog.openj9.org/2018/06/05/debugging-openj9-in-docker-with-gdb/). This will enable a minefield of assertions that will blow up if you nudge anything the wrong way. You'll love it. I also recommend setting a breakpoint at trclog.c:1448 to catch these and other assertions that may be triggered in OpenJ9 Java.

Appendix – Evacuator Tracing

Below is a scaled-down image of trace output for one evacuator GC cycle. It may be readable when scaled up.



```
  gc start; survivor{fa000000 100000000} tenure{d0000000 f4000000} evacuate{f4000000 fa000000} projection:100663296
    gc end; survivor:6913224 tenure:0 scanned:5485696; pulled:1602392; leaf:1427528; frag:904; realms:7.957; kb/ms:673.259; cache%:31.667; cpu%:385.736; yielded:2; switched:42
   objects: 0 0 0 0 79306 27499 7280 1394 552 585 135 546 21 25 17 10 117370 6913224
    arrays: 0 0 0 0 24793 4473 6739 1292 536 585 135 546 21 25 17 10 39172 4884112
copyspaces: 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 57 7737360
workspaces: 4 21 88 3 27 6 4 1 2 1 1 3 2 1 0 2 1 0 2 0 1 0 1 0 0 1 0 0 0 1 1 2 1602392 165
 work time; stall:85 0.035; wait:9 0.591; notify:16 0.031; sync:16 0.196; end:12 0.446; run%:395.413; cpu%:383.159; stall%:0.440; wait%:7.427; end%:8.068
 thread  0; scan:1; clear:2; stall:13; wait:1; run%:98.944; cpu%:95.400; stall%:0.176; wait%:1.571; end%:2.966
 thread  1; scan:1; clear:2; stall:62; wait:3; run%:99.007; cpu%:95.036; stall%:0.189; wait%:3.833; end%:1.206
 thread  2; scan:1; clear:2; stall:7; wait:5; run%:98.517; cpu%:95.111; stall%:0.038; wait%:2.023; end%:2.463
 thread  3; scan:1; clear:2; stall:3; wait:0; run%:98.944; cpu%:97.612; stall%:0.038; wait%:0.000; end%:1.433
    106660; 0 0 0 0
      1916; 1 0 0 0
      5109; 0 1 0 0
        35; 1 1 0 0
      2614; 0 0 1 0
       727; 1 0 1 0
       309; 1 0 1 1
conditions; so:1478 stf:161 st:54456 bfr:1109 rs:18189 sr:2697 sw:96160 sc:1109 io:39172 objects:117370
     11877; rs
       322; so rs
        20; so stf rs
       711; st rs
      1595; sr
        91; so sr
        18; so stf sr
     28152; sw
       339; so sw
        80; stf sw
     33826; st sw
       365; so st sw
        13; stf st sw
       324; st bfr sc
        53; bfr sw sc
       412; st bfr sw sc
      4734; rs io
       147; so rs io
         3; so stf rs io
       375; st rs io
       926; sr io
        57; so sr io
        10; so stf sr io
     14315; sw io
        20; so sw io
         3; stf sw io
     18175; st sw io
        86; so st sw io
         1; stf st sw io
       140; bfr sw sc io
        12; stf bfr sw sc io
       167; st bfr sw sc io
         1; stf st bfr sw sc io
```

The *gc-start* and *gc-end* rows should explain themselves. They are followed by two histograms *objects* and *arrays* counting evacuated objects (including all arrays) and pointer arrays by volume (in bytes) in bins with open endpoints at $\log_2(N)$, N<15 (the last bins are unbounded). The two large numbers at the end of these rows are the total evacuated object/array counts and volumes.

The *copyspaces* and w*orkspaces* histograms count refresh allocation and work release by volume in bytes in bins of equal size. The copyspace bins are 8196 bytes in size, workspace bins are 1024 bytes wide. The last number on the *copyspaces* row is the total evacuated volume. The last two numbers on the workspaces row are the total workspace volume and the number of workspaces pulled into evacuator scan stacks. The latter number may be smaller than the sum of the histogram bins because contiguous workspaces may be coalesced when pulled.

The *work-time* row presents aggregate thread statistics. The first six pairs of statistics are count and time in milliseconds spent in the respective state. The *run%* statistic is relative to the *realMs* reported in the *gc-end* row; it represents the percentage of 4**realMs* that evacuators were executing their main evacuation method, and c*pu%* represents the percentage of *run%* that evacuator threads were consuming CPU bandwidth. The per thread rows break down the *work-time* aggregated statistics by thread.

The 0/1 matrix following is an equivalence map on the set of evacuated objects induced by the controller's stalled thread map. Each row counts the number of objects evacuated by one of the 0s while the corresponding 1s were starved of scan work. This is followed by another equivalence map on evacuated objects (concordance) induced by object characteristics and operating conditions raised when each object was evacuated. The two-letter acronyms for these are described below (**leaf** may be added to this list in a later commit).

so	(stack overflow) forcing modal MICS (32) and MICD (0) and minimal workspace release (MWS) as stack winds down
stf	(survivor tail fill) forcing outside survivor copy to fill copyspace remainder
ttf	(tenure tail fill) forcing outside tenure copy to fill copyspace remainder
st	(stall) forcing modal MICS/MICD/minimal MWS to increase worklist volume
bfr	(breadth-first root) forcing outside copy for a root or remembered object
rs	(remembered set) this is raised while copying a remembered object (recursive scan unless **bfr** or object forced outside)
sr	(scan root) this is raised while copying a root object (recursive scan unless **bfr** or object forced outside)
sw	(scan worklist) this is raised while scanning the worklist (never with **rs** or **sr**)
sc	(scan clearable) this is raised during clearing stages (**bfr** is forced during clearing stages)
acp	(array copy) this is raised when copying a pointer array object
asc	(array scan) this is raised when scanning a pointer array object
bfa	(breadth-first always) forcing outside copy for all objects all the time (forces **bfr**)

A cursory look at the metrics for the sampled GC cycle show that the total evacuated volume (all survivor copy) was equal to the scanned volume (79% of copied volume) plus the volume of leaf objects (21%) dropped from the worklfow after copy, as expected. The controller asserts this invariant after each evacuation stage completes, and asserts that the total volume of survivor whitespace allocated equals volume used for copy pulls volume recycled or discarded at the end of each generational collection. The latter invariant my not hold for tenure whitespace because some it, sourced from one or more previous collections, is retained on tenure whitelists between collections until a global GC is triggered.

Of the volume scanned, 71% was copied inside, only 29% was copied outside and later pulled into Evacuator stacks from the worklists. Discovery of new scan work inline is a performance win for both Scavenger and Evacuator threads but inhibits production of distributable work. This can be seen by summing object counts for the stall map rows with >0 1s and comparing this sum with the sum of counts in the condition map for rows containing **st**. The stall map count will always be less than the **st** count because **st** is raised when an evacuator sees another evacuator stall *or* when its own worklist volume is below quota (typically 2*MWS). The magnitude of the difference (54% in the sampled GC) reflects how often evacuator threads are grinding with modal MICS/MICD and admitting primitive objects into outside copyspaces trying to pump up the volume of distributable scan work.