ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/306112816

Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK

Conference Paper · August 2016

DOI: 10.1145/2972206.2972210

CITATIONS
6

READS
7,193

5 authors, including:

Christine Flood
Red Hat Inc
18 PUBLICATIONS **534** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Shenandoah View project

All content following this page was uploaded by Christine Flood on 16 November 2017.

The user has requested enhancement of the downloaded file.

Shenandoah

An open-source concurrent compacting garbage collector for OpenJDK

Christine H. Flood
chf@redhat.com
Red Hat Inc

Roman Kennke
rkennke@redhat.com
Red Hat Inc

Andrew Dinn
adinn@redhat.com
Red Hat Inc

Andrew Haley
aph@redhat.com
Red Hat Inc

Roland Westrelin
rwestrel@redhat.com
Red Hat Inc

ABSTRACT

Shenandoah is an open-source region-based low-pause parallel and concurrent garbage collection (GC) algorithm targeting large heap applications. Snapshot At the Beginning Concurrent Marking and Brooks-style indirection pointer concurrent compaction enable significantly shorter GC pauses with durations that are independent of the application's live data size. Our implementation of Shenandoah in OpenJDK allows us to do comparison testing with mature production quality GC algorithms.

Modern machines have more memory and more processors than ever before. Service Level Agreement (SLA) applications guarantee response times of 10-500ms. In order to meet the lower end of that goal we need garbage collection algorithms which are efficient enough to allow programs to run in the available memory, but also optimized to never interrupt the running program for more than a handful of milliseconds. Shenandoah is an open-source low-pause time collector for OpenJDK designed to move closer to those goals.

CCS Concepts

•**Software and its engineering → Garbage collection;**

1. INTRODUCTION

1.1 The Problem

There are modern Java applications with 200gb heaps that are required to meet quality of service guarantees of 10-500ms. Compacting garbage collection algorithms have been shown to have smaller memory footprints and better cache locality than in place algorithms like Concurrent Mark and Sweep (CMS) [7]. Stopping the world to compact even 10% of a 200gb heap will exceed those pause time requirements. Meeting this level of service agreement requires a garbage collection algorithm which can compact the heap while the Java threads are running.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPPJ '16, August 29-September 02, 2016, Lugano, Switzerland

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4135-6/16/08...$15.00

DOI: http://dx.doi.org/10.1145/2972206.2972210

Stop the world (STW) garbage collectors (GCs) present the illusion to the mutator threads that objects are stationary. They stop the world, compact the live objects, and ensure that all references to moved objects are updated and then start the Java threads. By contrast, concurrent compaction requires that the GC moves objects while the Java threads are running and that all references to those objects immediately start accessing the new copy.

1.2 Contributions

The contributions of the Shenandoah algorithm are:

- Shorter pause times for OpenJDK due to concurrent compaction.
- Freely-available open-source implementation of a garbage collector with concurrent compaction.
- Architectural neutral algorithm, all that is required is a reliable atomic Compare and Swap (CAS) operation.
- Standard GC interface (Can compare performance of G1, CMS, Parallel, or Shenandoah)

2. THE IDEA

Concurrent compaction is complicated because along with moving a potentially in-use object, you also have to atomically update all references to that object to point to the new location. Simply finding those references may require scanning the entire heap. Our solution is to add a forwarding pointer to each object, and requiring all uses of that object to go through the forwarding pointer. This protocol allows us to move the object while the Java threads are running. The GC threads and the mutator threads copy the objects and use an atomic compare and swap (CAS) to update the forwarding pointer. If multiple GC and mutator threads were competing to move the same object only one CAS would succeed. References are updated during the next concurrent marking gc phase.

One of the goals of this project was that other GC algorithms would suffer no space or performance costs from having Shenandoah added to OpenJDK. This pure software solution doesn't change object layout. You may choose among various GC algorithms at run time. Heap walking tools will work. The compilers emit barriers only when running the Shenandoah collector.

The trade off is that Shenandoah requires more space than other algorithms. We could have chosen to use the mark word already present in Java objects in OpenJDK however that word is overloaded for many purposes including object locking. Checking and masking away those other uses would have made our read barrier significantly more expensive. A separate indirection word allows us to use a simple one instruction read barrier with minimal cost.

3. MAKING IT WORK

The idea is simple, but modifying a production quality compiler and run time system to use and optimize new read and write barriers was not. This paper will describe how we implemented the idea, the design decisions we made along the way, and the challenges we still face.

3.1 Object Layout

OpenJDK allocates two header words per object. One word is used to refer to the object's class. The other word, called the mark word, is really a multi-use word used for forwarding pointers, age bits, locking, and hashing. The object layout for Shenandoah adds an additional word per object. This word is located directly preceding the object and is only allocated when using the Shenandoah collector. This allows us to move the object without updating all of the references to the object. The thread that copies the object performs an atomic compare and swap on this word to have it point to the new location. All future readers/writers of this object will now refer to the forwarded copy via the forwarding pointer. If there is a race where one thread was reading the object at the same time as another thread was writing the object there's a slightly larger window for a race condition, but no new races were introduced.

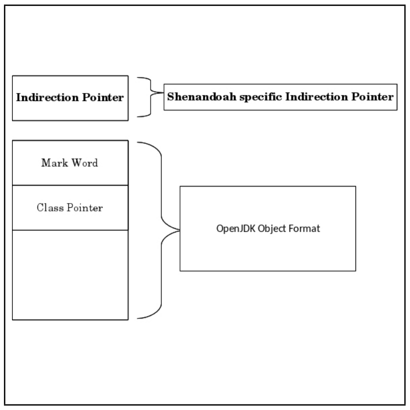

Figure 1: Shenandoah Object layout

3.2 Heap Layout

The heap is broken up into equal sized regions. A region may contain newly allocated objects, long lived objects, or a mix of both. Any subset of the regions may be chosen to be collected during a GC cycle.

We've developed an interface for GC heuristics which track allocation and reclamation rates. We have several custom policies to decide when to start a concurrent mark and which regions to include in a collection set. Our default heuristic which was used for our measurements chooses only regions with 60 percent or more garbage and starts a concurrent marking cycle when 75 percent of regions have been allocated.

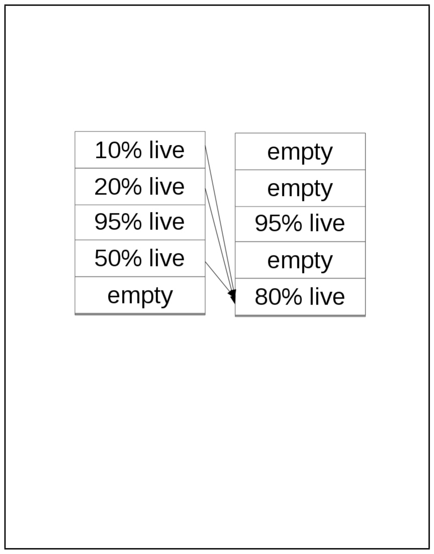

Figure 2: Shenandoah Heap layout

3.3 GC Phases

Shenandoah Phases

1. Initial Marking: Stops the world, scans the root set (java threads, class statics, ...)

2. Concurrent Marking: Trace the heap marking the live objects, updating any references to regions evacuated in the previous gc cycle.

3. Final Marking: Stops the world, Re-scans the root set, copy and update roots to point to to-region copies. Initiate concurrent compaction. Free any fully evacuated regions from previous compaction.

4. Concurrent Compaction: Evacuate live objects from targeted regions.

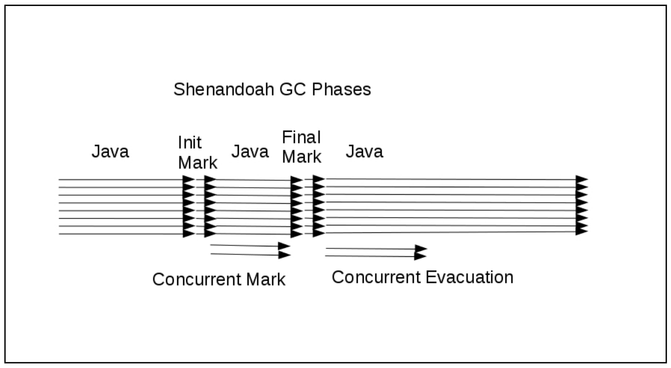

Figure 3: Shenandoah GC Phases

Each GC cycle consists of two stop the world phases and 2 concurrent phases. Please see figure 3 for a visual depiction of the phases.

We stop the world to trace the root set during an Initial Mark pause, then we perform a concurrent mark while the Java threads are running. We stop the world again for a final mark, and then perform a concurrent evacuation phase.

3.3.1 Concurrent Marking

We use a Snapshot At the Beginning (SATB) algorithm which means that any object that was live at the beginning of marking or that has been allocated since the beginning of marking is considered live. Maintaining an SATB view of the heap requires a write barrier so that if the mutator overwrites a reference to an object that object still gets traced during marking. We accomplish this by having a write barrier put overwritten values onto a queue to be scanned by the marking threads.

Each concurrent marking thread keeps a thread local total of the live data for each heap region. We combine all of these at the end of the final mark phase. We later use this data for choosing which regions we want in our collection set. Our concurrent marking threads use a work stealing algorithm to take advantage of as many idle threads as are available on the machine.

3.3.2 Choosing the collection set

We choose a collection set based on the regions with the least live data. Our goal is to keep up with mutation so we need to collect at least as much space as was allocated during the previous GC cyle. We have several available heuristics for choosing regions based on heap occupancy or region density. Pause times are proportional to the size of the root set, not the number of regions in the collection set, therefore we are free to choose a collection set based purely on collector efficiency.

3.3.3 Concurrent Compaction

Once the collection set has been chosen we start a concurrent compaction phase. The concurrent GC threads know how many bytes of data are live in the collection set, so they know how many regions to reserve. The GC threads all cooperate evacuating live objects from targeted regions. A mutator may read an object in a targeted region however a write requires the mutator also attempt to make a copy so that we have one consistent view of the object. It is an invariant of our algorithm that writes never occur in from regions.

The concurrent GC threads do most of the evacuation work using a speculative copy protocol.

- They first speculatively copy the object into their thread local allocation buffer.
- They then try to CAS the Brooks' indirection pointer to point to their speculative copy.
- If the CAS succeeds then they have the new address for the object.
- If the CAS fails then another thread beat them in the race to copy the object. The concurrent GC thread unrolls their speculative copy and proceeds using the value of the to-region pointer the other thread left in the indirection pointer.

When a Java thread wants to write to a from-space object it also uses the speculative allocation protocol to copy the object to it's own thread local allocation buffer. Only once it has a new address for the object may it perform the intended write. A write into a to-region object is guaranteed to be safe because to-region objects won't get moved during concurrent evacuation. We measured the amount of time the Java threads spent copying objects in write barriers when running the SpecJBB2015 benchmark and the time was negligible, less than 118 microseconds total over the two hour run.

Java threads may continue to read from-region data while other threads are copying objects to to-regions. Once the object is copied, the indirection pointer will point to the new copy and accesses will be directed there. By contrast GCs which require **all** accesses to be in to-regions may suffer from read storms. During evacuation forward progress may be significantly delayed by the mutator performing GC work to maintain the "all accesses must be in to-regions invariant". Our solution restricts this penalty to the normally far less frequent update case.

Note that reads of immutable data such as class pointers, array size, or final fields do not require read barriers because the value is the same in all copies of the object.

3.3.4 Updating References

Updating all of the references to from-region objects to point to their to-region copies requires a traversal of the entire heap. Rather than running this in a separate phase, and requiring stopping an additional time per gc cyle, we perform the updates during the next concurrent marking.

3.4 Barriers

Shenandoah relies on a read barrier to read through the Brooks' indirection pointer as well as a double write barrier. We have the SATB write barrier on stores of object references into heap objects. These object reference stores queue the overwritten values to maintain SATB correctness. We also have a concurrent evacuation write barrier which aids the concurrent GC by copying about to be written objects out of targeted regions to maintain our "no writes in targeted regions" invariant.

OpenJDK is a three tiered compilation environment. Methods that are only executed a few times are interpreted. Once a method hits a certain threshold of executions it is compiled using a text book style compiler called C1. Only when the method has hit an even higher threshold of executions will it be compiled using C2, the optimizing compiler. We implemented our read and write barriers for each tier of compilation, however the solutions are very similar. Here we will only discuss the optimizing compiler.

Barriers are required in more places than just when reading or writing the fields of objects. Object locking writes to the mark word of an object and therefore requires a write barrier. Anytime the VM accesses an object in the heap that requires a barrier.

3.4.1 Read Barriers

As shown in figure 4 our compiled read barriers are a single assembly language instruction.

```
void
ShenandoahBarrierSet::compile_resolve_oop(){
  __ movptr(dst, Address(dst, -8));
}
```

Figure 4: Shenandoah simple read barrier for fields of non-null objects

```
Here’s an assembly code snippet for
reading a field:

When we start register %rsi contains the address of the
object, and the field is at offset 0x10.

mov    0x10(%rsi),%rsi
   ; *getfield value


Here’s what the snippet looks like
with Shenandoah:

mov    -0x8(%rsi),%rsi
   ; read of forwarding pointer at address object - 0x8
mov    0x10(%rsi),%rsi
   ; *getfield value
```

Figure 5: Shenandoah read barrier snippets

3.4.2 Write Barriers

Write barriers need to do more work than read barriers but that is mitigated by their lower frequency. When concurrent marking is running we have an SATB write barrier which ensures that any values overwritten during concurrent marking are scanned by the concurrent marking thread. This barrier is exactly the same as the one used by G1 and CMS. Plus we have an additional Shenandoah specific write barrier which is only performed when we are in a concurrent evacuation phase which ensures that objects in targeted regions are forwarded before the write is attempted.

We employ an assembly level check to detect when we are in a concurrent evacuation phase, and if we are we call out to a runtime routine to copy the object when required.

The copying write barrier is required on all object writes including writes of base types and locking of objects. We are the only OpenJDK collector that requires write barriers on fields other than reference fields.

The order of the evacuation in progress check and the read of the indirection pointer is important. The evacuation in progress flag is set during the final marking STW phase, but cleared concurrently when all the copying work is complete. The flag may flip from true to false while we are in the write barrier. If we read the indirection pointer first, found the from-region object referencing it-self, and then read the false evacuation in progress field we might find ourselves writing to a from-region object. If we read the evacuation in progress flag first, than we are being conservative and are guaranteed to always call the write barrier when necessary.

3.5 if_acmpxx

Figure 7 shows you the issue with if_acmpeq and if_acmpne bytecodes. They compare two references. We initially believed that resolving both references would be sufficient, but unfortunately the GC may move one object out from under the mutator and cause false negatives. The naive solution would be to perform write barriers on both arguments, but we made that more efficient. We first perform a direct comparison of a and b and only if that fails do we then perform the comparison again with read barriers on both arguments

```
Here’s an assembly code snippet for
writing a field with Shenandoah:

0x00007fa8351e3f66: cmpb   $0x0,0x640(%r15)
   ; %r15 is the local thread
   ; 0x640 is the offset of the
   ;  evacuation_in_progress flag
   ; so we compare the evacuation_in_progress
   ; flag to zero
0x00007fa8351e3f6e: mov    -0x8(%r8),%r13
   ; Read the indirection pointer.
0x00007fa8351e3f72: je     0x00007fa8351e3f7f
   ; if !evacuation_in_progress jump to store
0x00007fa8351e3f74: xchg   %rax,%r13
   ; swap our object %r13 with %rax
   ; %rax is the expected input arg
0x00007fa8351e3f77: callq  Stub::shenandoah_wb
0x00007fa8351e3f7c: xchg   %rax,%r13
   ; swap the return value %rax
   ; which is possibly new address of
   ; our object back into %r13
0x00007fa8351e3f7f: mov    %sil,0x18(%r13,%rdx,1)
   ;*bastore {reexecute=0 rethrow=0 return_oop=0}
   ; - jdk.internal.org.objectweb.asm.ByteVector::
   ;         putUTF8@61 (line 255)
```

Figure 6: Shenandoah specific write barrier snippet

Figure 7: if_acmpxx problem

```
a’ = resolve(a);
gc thread moves a
b’ = resolve(b)
a and b are equal, but a’ and b’ aren’t.
```

. Read barriers are sufficient because the only way the a == b comparison will fail incorrectly is if one of the arguments has been copied.

Implementing Java level CAS of static and instance fields (under Unsafe or the java.util.concurrency classes) are handled similarly.

4. BARRIER OPTIMIZATIONS IN THE OPTIMIZING COMPILER

4.1 Safepoints

The Hotspot optimizing compiler issues safepoints at places where garbage collection events might occur. Places like method calls, and allocations. A final marking will occur at a safepoint and will copy all of the values referenced directly from the mutator threads and update the references in the threads before starting the concurrent evacuation phase. This means that we do not need to re-issue barriers after a safepoint.

4.2 Interference from other threads

The Java memory model allows us to eliminate redundant barriers. It’s OK for us to miss a write from another thread

Figure 8: if_acmpxx

```
if ((a != b) &&
   (resolve(a) != resolve(b)))
   return false
else
   return true
end
```

as long as there are no volatile accesses or memory barriers.

4.3 Barrier elimination

The read and write barriers are inserted at parse time. We created two new nodes in the intermediate representation for ShenandoahReadBarrier and ShenandoahWriteBarrier which are generated at the appropriate points during parsing. Global Value Numbering(GVN) is a phase in the optimizing compiler which assigns a value number to variables and expressions. If values have the same number then they are provably the same value. We've added code to the GVN phase to detect redundant barriers.

Rules for when read or write barriers are not needed:.

- If the value is newly allocated it will be in a to-region.
- If the value is a NULL pointer (NullPointerException).
- If the value is a constant.
- If the value comes from a write barrier.

Additionally read barriers may be eliminated:.

- If the value is guaranteed to be the same in both from and to-region copies (final fields).
- If the input value comes from a read barrier and there is provably no interfering write barrier.

Barriers may be hoisted outside of loops, and the optimizing compiler does that just as it would any other constant value.

4.4 Volatiles

Volatiles are treated like a full memory fence. They create a new memory state and any subsequent memory accesses are treated as new values.

5. DESIGN DECISIONS

5.1 Shenandoah isn't generational

Generational garbage collection algorithms focus their processing cycles on the youngest objects, based on the generational hypothesis that most objects die young. Some modern applications, such as web caches, hold onto objects just long enough to thwart generational garbage collectors. We wrote a small Least Recently Used (LRU) cache benchmark as an example of this behavior. The LRU benchmark models a URL caching program which maps web URLs to their corresponding content. We allocated 10,000 1.25mb binary trees and keep the most recently allocated 1000 in an array. We ran this program with both Shenandoah and G1 with a 4gb heap. Shenandoah was able to run this program in 16 seconds with 85 gc pauses taking a total of 4.24 seconds. G1 required 177 seconds, which is an order of magnitude worse than Shenandoah. This was entirely due to G1 having to execute 17 full gcs to keep up with the allocations, these full GCs took 105 seconds. These numbers look worse than they should because full garbage collections in G1 use a single threaded algorithm to compact the entire 4gb heap. The point is that Shenandoah was able to maintain a steady state, while the generational garbage collector continued to run out of space and be forced into performing full collections.

Generational garbage collection algorithms usually employ a card table to summarize references from old generation objects to young generation objects. This enables them to perform young generation collections and only scan/update the areas in the old generation which reference young generation objects. An implementation may accomplish this by summarizing the old generation with 1 bit in the card table for typically every 512 bytes of old generation data. This gives a compact representation of areas of the old generation which must be scanned and enables collecting only the young generation without scanning the entire old generation. The issue is that distinct areas of the old generation may actually require updates to the same cache line in the card table. This may degrade the performance of your carefully designed embarassingly parallel application.

Shenandoah will collect the regions with the most garbage, whether they are young or old. We do not employ a card table, so there are no surprise concurrency bottlenecks. The cost of the one word indirection pointer is slightly offset by no longer having an off heap data structure for managing old to young references.

6. HUMONGOUS OBJECTS

Programs sometimes allocate objects which are larger than a single heap region. If there isn't enough continuous space to allocate a humongous object then we will force a non-concurrent GC phase to try to make enough contiguous space. Humongous objects are never compacted. If a concurrent marking shows that a humongous object is no longer live than the heap regions it occupies may be immediately reclaimed. This means that we don't need to ever copy more than one object during a write barrier, and that object must be less than the region size. Humongous objects contribute to fragmentation. An object which occupies an entire region + 1 word, will take up two compete regions. It's possible to allocate objects in the remainder of that second region, but until the humongous object dies the second region will remain in use. It's also possible that humongous objects may fragment the heap enough that there are several small free spaces, but not enough contiugous free space to allocate a large object. It's possible to compact humongous objects by performing a full garbage collection, but we don't do that by default.

7. RESULTS

There was a performance degradation in many cases, however in all cases the amount of time in spent in GC pauses, and the length of the average and maximum GC pause de-

Table 1: SpecJBB

Algorithm	max-JOPS	critical-JOPS
Shenandoah	71652	43472
G1	80467	6199
Parallel	89190	19863
ParNew/CMS	43511	7355

Table 2: DaCapo 9.12 Benchmarks with no GC activity

Benchmark	Shenandoah	G1	percentage overhead
avrora	2096ms	2052ms	2.1 %
fop	1103ms	1044ms	5.6 %
luindex	861ms	832ms	3.4 %

creased significantly. Please keep in mind that these tests were run with the same size heaps regardless of the gc algorithm, which gives Shenandoah a handicap due to the additional space required for the indirection pointers. Shenandoah does not yet support compressed oops, so all runs were made with compressed oops turned off.

All tests were run on an Intel Brickland box running RHEL 7.

Intel Platform: Brickland-EX Cpu:Broadwell-EX, QDF:QKT3 B0 Stepping (QS), 2.2Ghz, 24 Core, 60MB shared cache COD ENABLED Intel(R) Xeon(R) CPU E7-8890 v4 @ 2.20GHz 524288 MB memory, 1598 GB disk space

7.1 SPECJBB2015

We ran SPECJBB2015 Composite with a 200gb heap using Shenandoah, G1, Parallel GC, and ParNew/CMS. SpecJBB measures Java server performance. It models a world-wide supermarket company and measures both critical throughput (Critical-JOps) under response time requirements as well as pure throughput (Max-JOps). Shenandoah's max-JOPS were within 20 percent of the parallel collector and performed better than ParNew/CMS. Shenandoah really shines in critical-JOPS where we received a score of more than twice the parallel collector. Please see table 1.

7.2 DaCapo

The DaCapo benchmarks [4] weren't really meant to measure server performance, especially with large heaps, but we've included them here for perspective. Figure 9 shows end to end run time of Shenandoah and the other OpenJDK garbage collectors.

Just for fun we ran a few of the DaCapo benchmarks with an obnoxiously large heap so that there was no GC activity. This measures just the added cost of our read/write barriers. As you can see in table 2 the barriers added only between 2.1 and 5.6 percent overhead. The other benchmarks were left out because their resource consumption triggered young generation GCs despite there being significant remaining heap.

7.3 ElasticSearch

We wanted to measure our performance with a benchmark that simulated a real customer application, so we downloaded approximately 200gb of wikipedia data from commoncrawl.org and indexed it using elasticsearch. We ran this benchmark with all of the OpenJDK GC algorithms with a 100gb heap. Shenandoah experienced a significant slowdown however our pause times were again much better. As you can see in Table 3 we had significantly fewer GC pauses than the other algorithms, and our pause times were measured in ms as opposed to seconds. We are investigating why our end to end run time was 24 percent more than the next closest algorithm.

8. LESSONS LEARNED

8.1 Debugging

One of the difficult lessons was that it's easy to miss a barrier. We found it helpful to mprotect pages of targeted regions and unprotect them only for the short intervals required to write a fowarding pointer. Mprotect is a linux system call which allows you to declare a page read only, and will signal a SIGSEGV if someone tries to write to the page. This technique was extremely helpful. Another technique for addressing this problem was adding a verification pass to the optimizing compiler. We walk the intermediate representation of the program graph to ensure that all nodes which perform reads or writes on the heap have a preceding barrier.

8.2 JNI Critical Sections

JNI Critical Sections are short sections of native (non Java) user code which must run uninterrupted by the JVM. Shenandoah can't enter a final mark pause until all JNI critical sections complete. Initial mark pauses don't move objects and therefore may run while the JNI critical section is running. When Shenandoah starts the final mark pause we check if there are any active JNI critical regions, if there are we toggle a switch and return immediately. That switch tells Java to block before entering any more JNI critical regions, and after the last Java thread leaves a JNI critical region trigger the final mark pause again. Other GCs need to wait for all objects to be evacuated before any JNI threads can enter critical regions, however we only need to wait until the final marking pause completes.

8.3 Weak References and Class Unloading

Weak references and class unloading are the cause of bulk of our final marking pause times. We can discover weak references concurrently, but we must process them during a stop the world pause in the order of strength. All strong references must be completed first, followed by soft, weak, phantom, and jni in that order. Theoretically class unloading could be done concurrently, but it's not straightforward and we haven't tackled it yet.

8.4 Locked Objects

Locking/Unlocking objects are writes and go through the write barrier protocol, however during out final-mark pause we eagerly evacuate all active monitors and this allows us to avoid a write-barrier on monitor-exit.

8.5 Time vs Space

Waiting to update references to evacuated objects until the next concurrent marking is efficient in terms of time spent, however it's very inefficient in terms of memory usage. The space overhead of Shenanodah is not just the forwarding pointer, but also those zombie regions which must be kept around for their forwarding pointers, and can't be

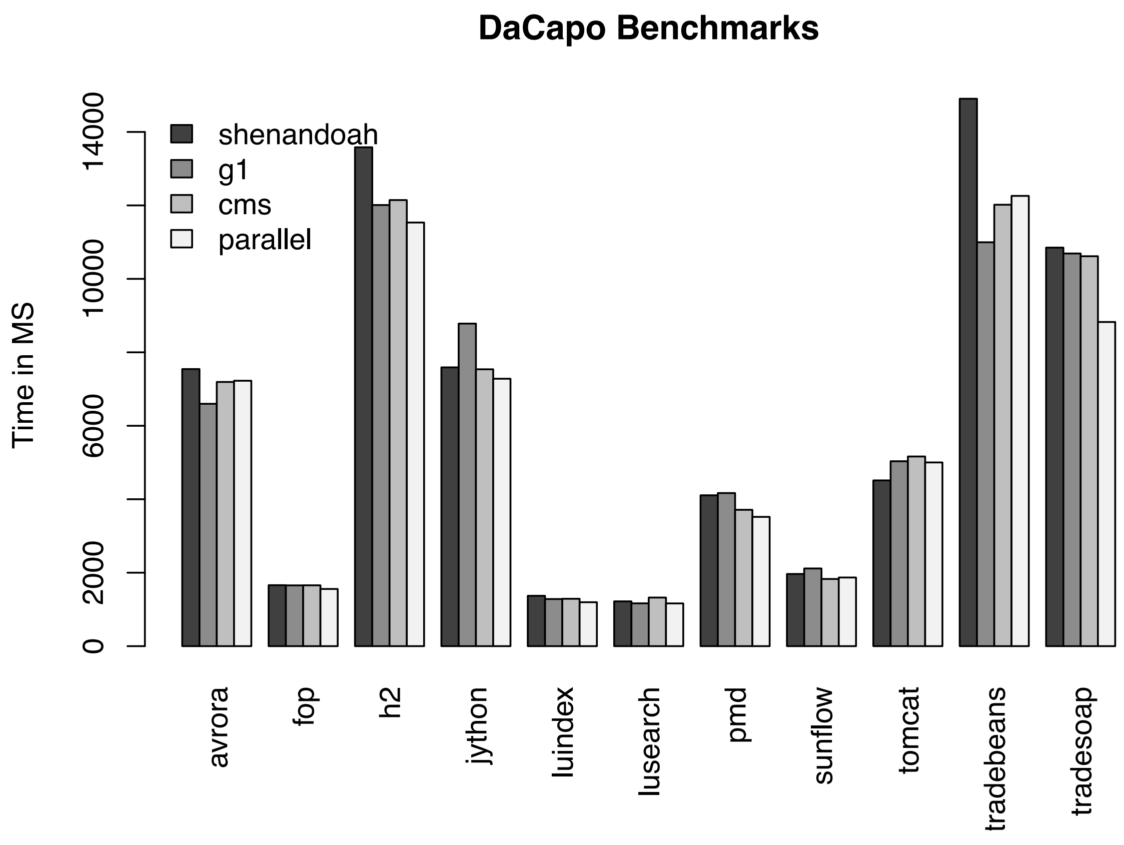

Figure 9: DaCapo Benchmarks

Table 3: ElasticSearch

Collector	Run Time	Total Pause	Max Pause	Avg Pause	Number of Pauses
Shenandoah	387.602s	320ms	89.79ms	53.01ms	6 (3 initial mark, 3 final mark)
G1	312.052s	11.7s	1.24s	450.12ms	26 (26 young, 0 old)
CMS	285.264s	12.78s	4.39s	852.26ms	15 (15 young, 0 old)
Parallel	260.092s	6.59s	3.04s	823.75ms	8 (6 young, 2 old)

freed until the next complete concurrent mark. There are strategies employed by other collectors [12] for dealing with this problem, but they rely on having a memory protection read barrier.

8.6 Stopping all of the threads

Shenandoah currently stops all of the Java threads to scan the thread stacks. We found that over the course of a SPECJBB run we spent 26.87 seconds in initial mark pauses, but only 10.53 seconds in our initial mark GC Code. We'd like to explore better methods of synchronization, as well as adopting some of the techniques from [9] for performing more of the work concurrently.

9. RELATED WORK

There are many different Garbage Collection (GC) algorithms available in OpenJDK. The Parallel garbage collector [8] stops the world to compact the heap but uses multiple work stealing threads [1] to get the work done quickly. Concurrent Mark and Sweep [11] runs while the Java program is running but doesn't perform compaction. G1 [7] is parallel and concurrent but still requires stopping the world to perform compaction. Shenandoah is parallel and concurrent and performs compaction while the Java program is running. Azul's pauseless [6] and C4 [12] collectors are parallel and concurrent and perform compaction while the Java program is running, however their algorithms aren't freely available in OpenJDK.

Our work is built on the early work of Brooks' incremental collector [5]. Metronome [2] [3] also uses a Brooks' style forwarding pointer however their work focuses on maintaining mutator usage guarantees. Shenandoah focuses instead on minimizing pause times.

There are many concurrent garbage collection algorithms in the literature. The most similar are the Chicken algorithm [10] and The Block-Free Concurrent G1 varient of Osterlund et al [9]. The Chicken algorithm requires a two instruction read barrier, one instruction to mask the tag bit, and one instruction to load the indirection pointer. Our algorithm only requires a single read. The Osterlund work uses a field pinning protocol to prevent the mutator and gc threads from accessing the same object at the same time. Both of these algorithms have a fallback position of not copying an object if a write conflict between the mutator and the gc exists. The Osterlund work [9] has given us much inspiration for further development of Shenandoah.

10. CONCLUSIONS AND FUTURE WORK

We have presented Shenandoah, a concurrent and parallel garbage collector which performs both the marking of the live objects, and the compacting of the heap concurrently with running Java threads. Shenanodah lowers the number of pauses as well as the duration of pause times and therefore increases the responsiveness of a Java Virtual Machine.

We have many plans to improve Shenandoah in the future. A NUMA version of Shenanodah would have separate work stealing queues for each NUMA node, and ensure data resides close to the thread that most recently accessed it.

Shenandoah currently uses a Snapshot at the Beginning (SATB) concurrent marking algorithm which ensures that all objects which were live at the start of concurrent marking are still live at the end of the concurrent marking. Incremental Update is a possible alternative which ensures that the concurrent marking algorithm traces all objects which are live at the end of concurrent marking. Any time the object graph is modified, the newly added values must be added to the marking queues. SATB was choosen for CMS and G1 because it has a simpler termination condition. CMS and G1 have young generation collections between concurrent markings, so having the concurrent marking algorithm identify young garbage isn't as important. . The benefit of catching short lived data earlier may be worth the more complicated termination protocol.

The Shenandoah heuristics we used in this paper are simplistic. We believe that there is more work to be done and with the correct heuristics the GC can be guaranteed to keep up with mutator allocation rates. Along the same lines we would like to study the mutator utilization guarantees of Shenandoah.

It's worth pursuing either a better way of stopping all of the threads or an algorithm which stops the threads one at a time as in [9]

Card Tables allow interim young generation collections without completing a full concurrent marking. Instead of breaking the heap into young and old regions and only allowing interim collections of the young regions, it should be possible to break the heap into connected subgraphs and perform an interim collection of an entire subgraph between concurrent markings.

10.1 Acknowledgments

The authors would like to thank the review committee especially Christian Wimmer for his help in reviewing this paper and the entire OpenJDK team at Red Hat for their help and support.

11. REFERENCES

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[2] J. Auerbach, D. F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, pages 245–254, New York, NY, USA, 2008. ACM.

[3] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[5] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 256–262, New York, NY, USA, 1984. ACM.

[6] C. Click, G. Tene, and M. Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 46–56, New York, NY, USA, 2005. ACM.

[7] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[8] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 21–21, Berkeley, CA, USA, 2001. USENIX Association.

[9] E. Österlund and W. Löwe. Block-free concurrent gc: Stack scanning and copying. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 1–12, New York, NY, USA, 2016. ACM.

[10] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 33–44, New York, NY, USA, 2008. ACM.

[11] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. In *Proceedings of the 2Nd International Symposium on Memory Management*, ISMM '00, pages 143–154, New York, NY, USA, 2000. ACM.

[12] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM.

View publication stats