Modern Application Specific Garbage Collectors

Garbage collection algorithms in recent times are getting popular. In modern times, garbage collection has provided reliability and productivity. In addition to these upsides, smart collectors have also assisted in achieving greater performances. However, garbage collectors are not universal. One certain collector will not work for all possible applications. Thorough and extensive inspection around the available collectors is mandatory to determine which collector is suitable for what application scenario and what application scenario favours which collector the most.

CCS Concepts: • **Programming language implementation → Garbage collection**;

Additional Key Words and Phrases: java, jvm, garbage collection, generational garbage collection, shenandoah, metronome, openjdk, memory management, real time collection, large heap size

ACM Reference Format:

https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Garbage collection is the automatic reclamation of space that has been allocated for objects previously but are no longer in use. Garbage collectors are very popular in recent times, and for good reasons. They provide excellent reliability as the programmer does not have to worry about memory management anymore. As a result, the abstraction provided to the high level programmers is better and they can put their effort in more complicated software engineering tasks. Automatic collection also ensures that memory is not wasted and smartly doing this makes the program achieve greater performances. However, this performance enhancement is not universal. A collector suited for application A may be catastrophic for application B. Therefore, careful application-specific inspection of the modern garbage collectors is of utmost importance.

Out of many important scenarios, two important ones are real-time systems, which require of a task to be completed before deadline, and applications that require very large heap. In this paper, we inspect three modern garbage collectors: ***Metronome***, ***Syncopation*** and ***Shenandoah*** which favor one of these two scenarios. These collectors vary in their nature of allocation, detection of garbage, collection, heap management, and overall implementation. We discuss in detail all these differences of these collectors throughout the paper.

The paper is organized as follows: in Section 2, we state our motivation behind this study. Next, in Section 3, we formally explain in detail the methodologies used in these collectors, in Section 4, we briefly describe the collector implementations in our own words, in Section 5, we present a comparative analysis of the mentioned collectors, in Section 6, we present some potential area of research around these, and finally, in Section 7, we finish with concluding remarks.

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 MOTIVATION

Garbage collection is an integral part of dynamic memory management in modern programming language implementations. Initially, this was made popular with Java. Automatically reclaiming the memory that is occupied by object which are no longer in use relieves programmers of manually managing memory. Therefore, garbage collection (GC) provides reliability and enhanced productivity.

With time, the research community realized that different application scenario demands different types of collectors. Real-time systems demand very predictable behaviour from the collector. Also, both in case of real-time and non real-time, searching the entire heap for these *garbages* is too much of a bottleneck, and therefore ***generational*** garbage collectors are popular. These collectors do not require of us to search the entire heap. Therefore, almost all the modern systems use generational collectors, and any collector which is not generational can hardly keep up with the generational ones in terms of performances in context of the modern programming language implementations. On the other hand, the application scenarios that work with a very large heap requires the collector to be parallel and the collector threads require more CPU time, regardless of the nature of the collector (generational or non-generational).

No class of collectors exists that are universally good for *all* application scenarios. For different scenarios, different collectors work good.

2.1 Garbage Collection

At its core, garbage collection [15] is a process of memory management so that someone as a developer have one less thing to worry about. When a program allocates memory - like by creating a variable - that memory is allocated to either the stack or the heap. Allocation is done in the stack when defining things in a local scope, such as primitive data types, arrays of a set size, etc. The stack is a self-managing memory store that we do not have to worry about - it is very fast at allocating and clearing memory all by itself. For other memory allocations, such as objects, buffers, strings, or global variables, allocation is done in the heap.

Garbage collection (GC) is a form of automatic memory management. The garbage collector, or just collector, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program. This memory (to be reclaimed) is allocated in the heap. Garbage collectors do not worry about the stack, as it is self-managing.

Garbage collection was invented by **John McCarthy** around 1959 to simplify manual memory management in Lisp [14]. McCarthy was also the first to use the term *garbage*.

Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of approaches, including other techniques such as stack allocation and region inference. Like other memory management techniques, garbage collection may take a significant proportion of total processing time in a program and, as a result, can have significant influence on performance. With good implementations, enough memory, and depending on application, garbage collection can be faster than manual memory management, while the opposite can also be true and has often been the case in the past with sub-optimal GC algorithms.

2.2 Real-Time

A real-time system is a hardware or software system that is subject to deadlines from event to system response. Some soft real-time systems such as video displays can tolerate the occasional failure; a dropped frame is not the end of the world. There are also hard real-time systems such as those running internal combustion engines, which rightly consider even one missed deadline

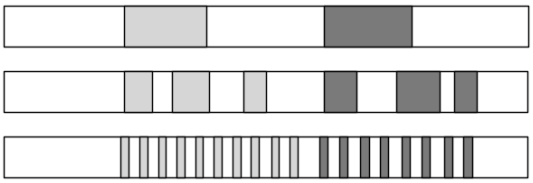

Fig. 1. Mutator (white) and Collector (non-white) timings during normal execution and two GC runs. Top: Stop the World, Middle: Incremental, Bottom: Real-Time

a total system failure: missing a deadline can mean damaging an engine. What is important in real-time systems is that the application must be guaranteed to almost never miss a deadline. Therefore, real-time garbage collection, or simply *RTGC* requires that the collector behaviour be very predictable. The requirement is explained in Figure 1 [3].

The top band shows two long GC pauses which reflect normal behavior of Stop the World GC such as vanilla Mark-and-Sweep. While the pauses appear consistent, they are each too long. For this reason, RTGC is not often pursued in a non-incremental context: it is too difficult to keep pause times below a desired threshold. The middle band, which shows shorter but more irregular GC pauses over two collections, illustrates a normal incremental GC such as a Baker copying collector. While the lengths are better, the pauses are simply too irregular. Lastly, the bottom band above shows ideal behavior for RTGC. Pauses are short and regularly spaced out - in a word, predictable.

In short, Real-Time Garbage Collection is Automatic Memory Management capable of:

- Guaranteeing a certain amount of minimum mutator utilization (***MMU***) in a given time window
- Accounting precisely for all mutator interruptions
- Ensuring that space bounds are not exceeded

2.3 Generational

Generational GC is based on the hypothesis: "Most objects die young". Analyses have found that almost 70% of young objects do not survive and become garbage immediately [17]. Therefore, whereas we know that around 70% of the newly allocated objects will be garbage, it is not very efficient to search the entire heap for these *new garbages*. Therefore, we divide the heap into two spaces - ***The Nursery Space*** and ***The Mature Space***.

When allocated, we allocate new objects in the nursery space. We run GC algorithms in the nursery space only. When a certain time passes, the objects are promoted from nursery space to the mature space. And finally, when there is no free space in the nursery, then and only then, we run a full heap garbage collection.

It is very important to note that the heap size for certain set-ups is very large, maybe 2-4 GBs. Therefore, searching the entire heap to detect the garbage objects in every GC cycle sill halth the application thread(s) much too often, and therefore deteriorate the overall performance. Generational GC relieves the collector of looking at the entire heap. Therefore, for the modern applications, it is almost always a generational collector in play, and almost always the generational collector outperforms the other collectors.

2.4 Concurrent

Concurrent garbage collectors are those where collector and mutator threads work in parallel. Whenever the workload on the collector is very large, these collectors are particularly very useful.

However, achieving concurrency of mutator and colletor is very difficult and introduces additional bottlenecks.

3 METHODOLOGY

In this section, we are going to describe three modern garbage collectors: ***Metronome***, ***Syncopation*** and ***Shenandoah***. These are very recent implementations of collectors with excellent performances. They vary in their styles of methodology, idea, implementation and intended application scenario.

3.1 Metronome

Metronome is a real-time garbage collector. It was originally intended for processing real time audio, mainly in IBM Research. Before Metronome, There have been early attempts at providing real-time [8] or, at least, suitably responsive [18] garbage collectors to various environments. However, these systems were either impractical because of a flawed definition of real-time or the response was only often but not always acceptable.

3.1.1 Overview. Researchers at IBM Research have developed a hard real-time garbage collector called Metronome, first as a research prototype [7] and then as the central component of a new real-time Java virtual machine product. The technology has been adopted for time and safety-critical systems by companies in a number of industries, including its use by Raytheon for the development of the software for the next-generation Navy destroyer.

The real-time Java virtual machine runs on top of a customized Linux kernel, co-developed by IBM and the open-source community. These modifications to Linux are making their way into the mainstream, with most of them being incorporated into the real-time edition of Redhat Enterprise Linux 5 (RHEL5 RT). The combination of these technologies raises the possibility of programming hard real-time systems on commodity hardware using a widely-distributed open-source operating system and a highly portable, high-level object-oriented language.

In the work described in the original paper of Metronome [2], researchers set out to systematically investigate the capabilities and limits of our technology in the domain of interactive music synthesis. Finally they built a music synthesizer from scratch, entirely in Java. They made extensive use of Java's object oriented features, including frequent dynamic allocation of objects. This allowed them to significantly simplify and yet also generalize the system, relative to previous experience implementing synthesizers in C and C++.

3.1.2 Metronome Garbage Collector. Metronome [7] is a garbage collector designed to support hard real-time, with highly predictable latencies down to the millisecond level. It is a key part of the IBM WebSphere Real Time product. Its technology differs from earlier approaches to "real-time" collection in several fundamental ways.

First, Metronome's approach to collection allows virtually all of the collection work to be done asynchronously by the collector. This is in contrast to previous approaches, which often required the application threads to perform work (like copying objects or updating pointers) on behalf of the collector, causing uneven and unpredictable performance of the application threads.

Second, Metronome takes advantage of this decoupling and uses *time-based* scheduling of garbage collector work, where the collector (when it is active) runs at regularly scheduled intervals of regular sizes. This makes the impact of the collector on the application highly predictable and reliable. By contrast, previous approaches typically performed work-based collection, where a unit of collector work was performed in response to application activity, for instance for every 4KB of memory allocated. Since allocation is typically unevenly distributed, *work-based* scheduling results in uneven interruptions of the application.

Third, Metronome is engineered in such a way that its individual work quanta are extremely small, both in the average and worst case, to the point where its resolution is sufficiently high for the vast majority of real-time applications.

Metronome's time based scheduler operates by providing a minimum mutator utilization or MMU [10]. The MMU specifies a time window and a percentage of the CPU time within that window which must be allocated to the application (a.k.a. the mutator). For instance, an MMU of 70% with a 10 ms time window means that for any 10 ms time period in the program's execution, the application will receive at least 70% of the CPU time (i.e., at least 7 ms).

A naive application of this approach would result in a maximum pause time of 1 *MMU* times the window size (e.g., 3ms in the example). Also, the MMU contract is vulnerable to overshooting if the collector mis-estimates the time for a piece of work. This is solved by *overclocking* the collector scheduler. The collector runs at a nominal quantum size of 500 μs, and those quanta are spread out over the MMU window. Thus for the example above, we would nominally perform 6 separate quanta to perform the 3 ms of collector work. Those quanta are evenly spread, so that the application experiences less and more evenly distributed collector-induced jitter. Furthermore, the scheduler is able to do a much better job of guaranteeing the MMU target, since if a single quantum is larger than expected, it can simply either delay the next quantum slightly, or else perform a shorter quantum the next time.

In order to bound the work of scanning or copying an array (one aspect of bounding collector pauses), and also to bound external fragmentation caused by large objects, Metronome relies on *arraylets*, a technique for breaking large arrays into chunks that can be processed in bounded time. In the arraylet object model, all arrays consist of two parts: a spine, which contains the object header and pointers to arraylets, which are contiguous fixed-size chunks. Using 2^n sized chunks allows the indexing calculation to be done with shift and mask operations.

There are other analyses in the original paper of Metronome as well. However, those correspond to the audio processing capabilities of the system that has been developed and bear little correspondence to garbage collection. Hence, we do not present those here.

3.2 Syncopation

Metronome [2] is the first hard real-time garbage collector. It was the first guaranteed hard real-time collector. This collector provided guaranteed MMU based on the the characterization of the application in terms of maximum live memory and allocation rate. Space overhead was usually comparable to that required by synchronous ("stop-the-world") collectors, due to incremental defragmentation and quantitative bounding of all sources of memory loss [6]. However, The result was a collector that is able to guarantee a minimum mutator utilization of 50% at a resolution of 10ms: out of every 10ms, the mutator threads receive no less than 5ms - with no exceptions. During periods when collection is off, the mutators receive almost all of the CPU (a small portion is charged to the collector for things like allocation operations). Collection is active about 45% of the time, resulting in good but not exceptional throughput.

With rise of Metronome, there was significant interest from potential users of this technology. There were two major barriers to the adoption of our system, one technical and one practical. The technical problem was that a utilization level of 50% during collection was not acceptable to some users: they wanted more CPU time available for real-time tasks even while garbage collection was active.

The second barrier to the adoption of the system was practical: they required a complete Java development environment with full library and debugging support. The Metronome was implemented in Jikes RVM [1], which lacks these attributes.

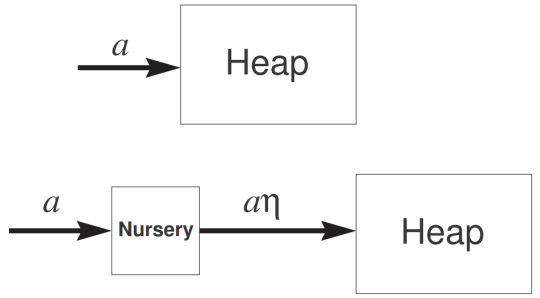

Fig. 2. For a program with allocation rate a, interposing a nursery reduces the effective allocation rate to $a\eta$, where η is the nursery survival rate.

Therefore, addressing these issues, mostly the same team of researchers from IBM Research worked to build a collector with the same predictability as the Metronome, but with higher utilization and increased throughput, which is called Syncopation [5]. To achieve higher utilization, they used generational collection [16], which focuses collector effort on the portion of the heap most likely to yield free memory. This has added benefits in a concurrent collector because it reduces the amount of floating garbage, which is typically a drawback of such systems. However, collecting the nursery is unpredictable both in terms of the time to collect it and the quantity of data that is tenured.

Before going into the details of their contributions, it is worth mentioning their goal formally. The goal is meeting real-time deadlines, with high utilization. Say the real-time deadline is ΔT. That means, withing a time window of ΔT, we must allow for the mutator to run at least $u\Delta T$ time. If we cannot guarantee that, then the system will not be able to meet the real time requirements and thus, fail.

3.2.1 Introducing Generational Collection in Metronome. In some cases the Metronome may not be able to meet an application's real-time utilization requirements. In that case, there are a number of things the programmers can do: they can increase space consumption by buying more memory; they can decrease the utilization requirement u by buying a faster processor; they can reduce the allocation rate a by rewriting the code to perform less dynamic allocation; or they can reduce the live memory size by rewriting the code to reduce the size of long-lived data structures.

However, to the greatest possible extent we wish to avoid placing such a burden on the user. Within the system, there are several ways to improve matters: speed up the collection rate R by tuning the collector; decrease m by using various compaction techniques; or decrease the fragmentation factor ρ by improving the heap architecture and free space management.

Unfortunately none of these techniques is likely to provide the order-of-magnitude improvements that we require. The original Metronome collector had already been fairly well tuned; fragmentation was bounded fairly tightly (12% in theory and 3% in practice); and they have already applied object model compression techniques to the J9 virtual machine [4] in which they implemented the collector described in this paper.

However, there is a way in which we can reduce the allocation rate a. If we employ generational techniques, we can view the nursery as a filter which reduces the allocation rate into the primary heap to $a\eta < a$, as shown in Figure 2. In general we expect $\eta \ll 1$which will greatly reduce the load on the main collector. Of course, collecting the nursery will also have a cost, but for most applications we expect that the benefits of reducing the workload on the main collector will normally outweigh these costs by a large margin.

3.2.2 Synchronous Nursery Collection. Typical stop-the-world generational collectors consist of two disjoint collectors: one for the nursery and the other for the tenured (heap) space. Both collectors are usually run with mutators stopped. In a scheme where heap collection is incremental, care must be taken to synchronize with nursery reclamation. Moreover, the incremental collector requires the use of a snapshot write barrier. Therefore, the write barrier must provide both generational and snapshot functionality.

Nursery collection is typically triggered when the nursery is full. In an incremental system mutators can interleave with the collector. Therefore a mutator can evacuate the nursery either when the collector is not running or when the collector is marking or sweeping. In addition, the incremental collector itself evacuates the nursery at the beginning of its root scanning phase. There are three optimizations arising from this step. First, the snapshot write barrier does not need to record the overwriting of nursery to heap pointers. Secondly, during its heap marking phase, the incremental collector does not need to trace nursery objects. Thirdly, we can make sure that we eliminate all of the floating garbage in the nursery. If we choose not to evacuate in the beginning of our collection cycle, then the above optimizations cannot be applied. The incremental collector also performs nursery evacuation at the start of its sweeping phase.

Aside from when nursery evacuation occurs, another effect of combining generational and incremental collectors is the write barrier operation. Although both write barriers protect against the loss of a reachable object, the snapshot and the generational barrier share the following fundamental differences:

- Generational barriers are *always* active, snapshot barriers are *partially* active: only when the collector is heap marking.
- Generational barrier entail object/region rescanning for pointer fixup, snapshot barriers do not.
- Generational barrier remembers the destination object/region, snapshot barriers remember the overwritten pointer.
- Generational barrier performs range comparisons to determine whether the new pointer is a heap to nursery one, the snapshot barrier does not perform range checks.

Most pointer stores are nursery to nursery pointers. Since nursery collection is synchronous, we do not require a snapshot write barrier on those pointers. Additionally, since we evacuate the nursery at the start of root set scanning, we also do not need to barrier nursery to heap pointers. Therefore, the common write barrier can now filter on the destination object. That is, if the destination object is in the nursery, we do not need either of the two barriers. For snapshot, we also do not need to barrier heap to nursery pointers, which are needed by generational barrier. Conversely, the snapshot barrier needs to record heap to heap pointers while the generational barrier does not.

Therefore, with this new insight, we can utilize the range checks that the generational write barrier performs on the destination object in order to to filter out significant number of snapshot barriers. We note how write barrier operation is connected with the timing of the nursery evacuation.

In a real-time environment, if we are performing synchronous nursery collection then we must be able to compute the worstcase execution time (WCET) for nursery collection. This means carefully bounding all possible sources of work. In particular, the remembered set is also allocated from within the nursery. Objects are allocated left to right, and remembered set entries from right to left. When the two regions meet, the nursery is full and must be collected.

3.2.3 Handling Temporary Spikes in Allocation Rate: Syncopation. The real-time constraint is all about meeting the deadlines and ensuring that the utilization *never* drops below the desired

level. Therefore, temporary spikes in the allocation is of particular interest. Because if due to these spikes, we are unable to meet the deadline even once, the system will consequently fail.

There are two basic approaches that has been undertaken for *Syncopation*, which means a temporary displacement of the regular metrical accent in music. They are:

- ***Syncopation via Allocation Control*** In this one, we dynamically resize the nursery (virtually, not physically) to ensure that we can still perform a worst-case evacuation of the nursery without violating the mutator utilization requirement. For example, if $\Delta t = 10ms$ and $u = 0.7$ and if the first nursery collection in the interval consumes 2 ms, then we can still consume 1 ms for collection. If we resize the nursery so that its collection has WCET=1 ms, then the system can proceed safely.
  When the nursery becomes too small to be useful, we change allocation policy and simply allocate all objects directly in the heap; we call this *floodgating*. During such a period, the effective allocation rate into the heap will spike from the nursery-attenuated rate aÎů to the full rate a. The system continues to dynamically monitor the MMU, and when it has risen sufficiently it switches back to a nursery allocation policy.
  The advantages of this approach are that it is effective and relatively simple to implement. The disadvantages are that it requires provisioning for the worst case in advance, which means that it can not make full use of available processing resources, and that it can only adapt to variations in a, but not in η. Furthermore, it requires a conditional on the critical path of the inlined allocation sequence, which slows down all allocations.
  To understand why floodgating precludes the full use of an available collector quantum, consider the case without floodgating in which we consume a full collector quantum $(1-u)\Delta t$ to collect the nursery. It is very desirable to be able to do so since it means that we can use the largest possible nursery, which will maximally attenuate η.
  However, when we finish collection we have used our full quantum, so we must guarantee that the mutator will run for at least $u\Delta t$ before any collection takes place. However, there is no bound on the instantaneous allocation rate of the mutator. It could fill the nursery within $u\Delta t/2$ time units, at which point we would be forced to synchronously evacuate the nursery and would fail to meet our MMU commitment.
  The only alternative is to immediately begin tenuring newly allocated objects and not allow any nursery allocation for $u\Delta t$ time. At that point, we perform another collection quantum, and are back to where we started: being unable to allocate into the nursery.
- ***Syncopation via Collection Control*** The limitations of allocation control give rise to an alternative method of syncopation based on controlling the collection, rather than the allocation. In this regime, the collector begins collecting the nursery at the beginning of the collector quantum. If it completes collection in time, it resets the nursery and resumes the mutator. However, if the collector quantum expires before nursery collection is complete, it syncopates: it unconditionally tenures the remaining unevacuated objects by logically moving the nursery into the mature space. This is done by appending it to a list of *off-beat* pages. Then a new nursery is obtained from a pre-allocated reserve and the remembered set buffer is reset. Since all these operations are done logically, by redirecting pointers, syncopation is extremely fast.
  This method of syncopation is made possible by the presence of a read barrier in the collector. As the nursery is being collected and objects are moved into the heap, the forwarding pointer that is left behind has the exact same format as the forwarding pointer used to facilitate incremental object movement in the main heap. Therefore, when a partially collected nursery is tenured, heap to nursery references that were stored in the remembered set but not yet

fixed will simply follow the forwarding pointer via the read barrier.
The major advantage of collection-based syncopation is that it allows the collector to consume a full mutator quantum, without requiring a change in the allocation policy. If the allocation rate subsides in the subsequent quantum, the collector will immediately regain the full benefits of generational collection. The primary disadvantage is fragmentation: even if only one object in the nursery is live, none of the memory will be reclaimed until a full collection has taken place. Even worse, since the nursery was allocated sequentially rather than using segregated free lists, it must be completely evacuated before it can be reclaimed, increasing the defragmentation load and making the memory unavailable for one and a half major collection cycles (one to move the live objects, the other to forward any pointers to them).
In the limit, collection-based syncopation degenerates into a semi-space collector, in which all nurseries are syncopated and become the from-space. However, it would be almost impossible to cause this to happen, even with an adversary program.
While syncopation makes it possible to use a full collector quantum for collection, and to allow the nursery to grow to a point beyond its WCET collection limit, it is undesirable to do so. Thus it may still be desirable to have multiple collector beats per measure, as in allocation-based syncopation, although we expect that significantly fewer beats will be required.

3.3 Shenandoah

Shenandoah [12] is an ultra-low pause time garbage collector that reduces GC pause times by performing more garbage collection work concurrently with the running Java program. CMS and G1 both perform concurrent marking of live objects. Shenandoah adds concurrent compaction, which means its pause times are no longer proportional to the size of the heap. Garbage collecting a 100 GB heap or a 2 GB heap has the same predictable pause behavior.

3.3.1 Overview. 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) [11]. 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.

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.

Identifying the problem, researchers from Red Hat Inc. have developed ***Shenandoah***, solving these potential problems.

3.3.2 Complication Associted with Concurrent Compaction. 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

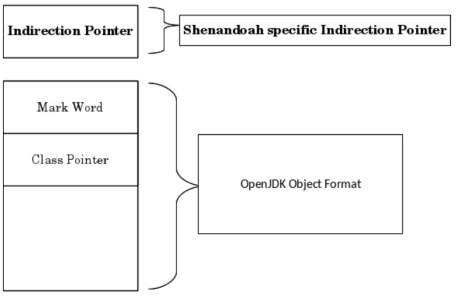

Fig. 3. Shenandoah Object Layout

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 is designed to work for applications that require large heaps, and so Shenandoah itself 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.

3.3.3 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.

3.3.4 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.

They have 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.

3.3.5 GC Phases. Shenandoah is the regionalized collector, it maintains the heap as the collection of regions. The regular Shenandoah GC cycle looks as shown in Figure 5.

The phases above do roughly as follows:

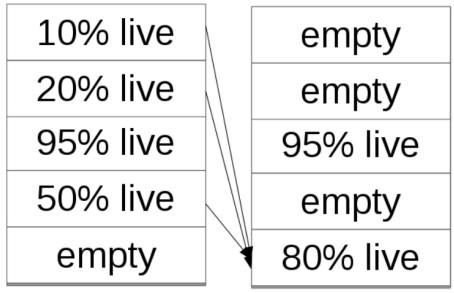

Fig. 4. Shenandoah Heap Layout

- **Init Mark** initiates the concurrent marking. It prepares the heap and application threads for concurrent mark, and then scans the root set. This is the first pause in the cycle, and the most dominant consumer is the root set scan. Therefore, its duration is dependent on the root set size.
- **Concurrent Marking** walks over the heap, and traces reachable objects. This phase runs alongside the application, and its duration is dependent on the number of live objects and the structure of object graph in the heap. Since the application is free to allocate new data during this phase, the heap occupancy goes up during concurrent marking.
- **Final Mark** finishes the concurrent marking by draining all pending marking/update queues and re-scanning the root set. It also initializes evacuation by figuring out the regions to be evacuated (collection set), pre-evacuating some roots, and generally prepares runtime for the next phase. Part of this work can be done concurrently during Concurrent precleaning phase. This is the second pause in the cycle, and the most dominant time consumers here are draining the queues and scanning the root set. Final mark phase also reclaims immediate garbage regions - that is, the regions where no live objects are present - this is why heap occupancy drops down at this phase.
- **Concurrent Evacuation** copies the objects out of collection set to other regions. This is the major difference against other OpenJDK GCs. This phase is again running along with application, and so application is free to allocate. Its duration is dependent on the size of chosen collection set for the cycle.
- **Init Update Refs** initializes the update references phase. It does almost nothing except making sure all GC and applications threads have finished evacuation, and then preparing GC for next phase. This is the third pause in the cycle, the shortest of them all.

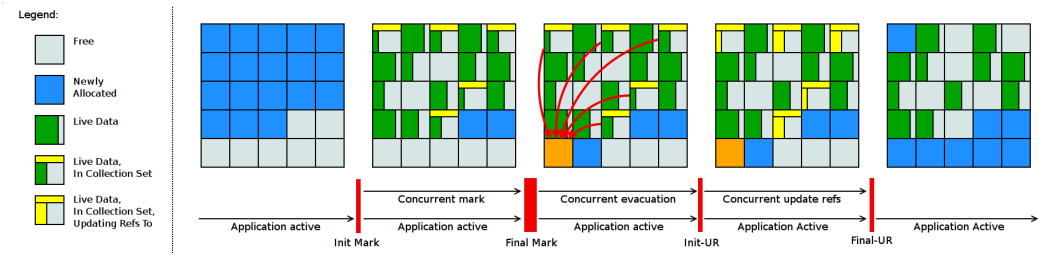

Fig. 5. Shenandoah GC Cycle

- **Concurrent Update References** walks over the heap, and updates the references to objects that were moved during concurrent evacuation. Its duration is dependent on number of objects in heap, but not the object graph structure, because it scans the heap linearly. This phase runs concurrently with the application.
- **Final Update Refs** finishes the update references phase by re-updating the existing root set. It also recycles the regions from the collection set, because now heap does not have references to (stale) objects to them. This is the last pause in the cycle, and its duration is dependent on the size of root set.

Root set includes thread local variables, references embedded in generated code, interned Strings, references from classloaders (e.g. static final references), JNI references, JVMTI references. Having larger root set generally means longer pauses with Shenandoah, see below for diagnostic techniques.

3.3.6 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.

3.3.7 Performance Dependencies of Shenandoah. Shenandoah performance, like the performance of almost all other GCs, depends on heap size. We expect it to perform better in cases when there is enough heap space to accommodate allocations while concurrent phases are running. The time for concurrent phases correlates with the live data set size (LDS) – the space taken by live data. Therefore, the reasonable heap size is dependent on LDS and allocation pressure in the workload: for a given allocation rate, larger LDS-es require proportionally larger heap sizes; for a given LDS, larger allocation rates require larger heap sizes. For some workloads with minuscule live data sets and moderate allocation pressure, 1...2 GB heaps performs well. We routinely test on 4...128 GB heaps on various workloads with up to 80% LDS size.

3.3.8 Heuristics Used in Shenandoah. Heuristics tell when Shenandoah starts the GC cycle, and regions it deems for evacuation. Heuristics can be selected with -XX:ShenandoahGCHeuristics=<name>. Some heuristics accept configuration parameters, which might help to tailor the GC operation to your use case better. Available heuristics include:

- **Passive:** This heuristics tells GC to be completely passive. Once available memory runs out, Full Stop-The-World GC would be triggered. This heuristics is used for functional testing, but sometimes it is useful for bisecting performance anomalies with GC barriers (see below).
- **Aggressive:** This heuristics tells GC to be completely active. It will start the new GC cycle as soon as the previous one finishes, and it will evacuate all live objects. This heuristics is useful for functional testing of the collector itself. It incurs heavy performance penalty, because GC is actively stealing lots of CPU cycles from the application.

- **dynamic:** This heuristics decide to start GC cycle based on heap occupancy and allocation pressure. Useful tuning knobs for this heuristics are:
  - -XX:ShenandoahFreeThreshold=#: Set the percentage of free heap at which a GC cycle is started
  - -XX:ShenandoahAllocationThreshold=#: Set percentage of memory allocated since last GC cycle before a new GC cycle is started
  - -XX:ShenandoahGarbageThreshold=#: Sets the percentage of garbage a region need to contain before it can be marked for collection
- **Adaptive (default):** This heuristics observes the previous GC cycles, and tries to start the next GC cycle so that the free space available at all times was below the "free threshold". Useful tuning knobs are:
  - -XX:ShenandoahInitFreeThreshold=#: Initial remaining free threshold
  - -XX:ShenandoahMinFreeThreshold=#: Minimum remaining free threshold
  - -XX:ShenandoahMaxFreeThreshold=#: Maximum remaining free threshold
  - -XX:ShenandoahHappyCyclesThreshold=#: How many successful marking cycles before improving free threshold
  - -XX:ShenandoahGarbageThreshold=#: Sets the percentage of garbage a region need to contain before it can be marked for collection.

In some cycles, Update References phase is merged with Concurrent Marking phase, at heuristics discretion. You can forcefully enable/disable Update References with *-XX:ShenandoahUpdateRefsEarly = [on|off]*.

4 DIFFERENT TECHNIQUES

In this section, we will describe the insights of the three aforementioned garbage collectors.

4.1 Metronome

As mentioned earlier, Metronome is a real-time collector that guarantees meeting the real-time deadline. This is the first collector that is able to do such claim. They guarantee a certain utilization u of the mutator within a real-time deadline window Δt. That means, the mutators are guaranteed to run at least $u\Delta t$ time.

The creators of metronome have achieved this through a very interesting approach - ***time based scheduling***, instead of traditional ***work-based scheduling***. Work based scheduling means scheduling a garbage-collection cycle when certain thing has happened, *not* when a certain time has elapsed. When the scheduling is performed in a work-based manner, we cannot ensure that the collector will run for this period of time. However, when the scheduling is time based, this can be said. Given that the application has been characterized good enough by the user, a certain amount of utilization can be guaranteed, both theoretically and practically, which is exactly what was presented in the paper of Metronome.

The naming of the collector is also interesting in the nature. The paper describes the integration of the Metronome collector with real-time music processing. Metronome is a musical instrument (now-a-days, a simple desktop or web application). Beginner musicians set a certain beat-rate in a metronome, and the app plays beats in that beat-rate. It is almost as if a drummer is drumming in a certain beat-rate. With that beat-rate, the beginner musicians practice various instruments. It is quite common amongst stringed instrument players, like cello, violin, guitar. They play along with a metronome set at certain beat-rate, and thus practice to be able to synchronize their own playing with other instruments. Simply googling “metronome” also gives a standard metronome that can play from 40 to 218 bpm (beats-per-minute).

Now, our collector, *Metronome* is named so because it alternates between the collector and the mutator in a regular, uniform, timely fashion. It guarantees to switch back to the application thread, within a certain time window, as a metronome plays beats in a regular fashion.

Other than this alternating between application and collector, Metronome has other important features. One of them is that the *entire collection is now performed by the collector*. Implementation of various schemes sometimes employ that the or the mutator thread(s) performs some of the activities of garbage collection. However, in a real-time system, the utilization is very important and the application must be given as much CPU cycles as possible, hence the collection work is entirely given to the collector.

Another important feature is that individual work quanta of Metronome are very small, both in the average case and in the worst case. As a result, the behaviour of the collector thread is very predictable. Therefore, unpredictable collector overheads are totally minimized.

Metronome uses ***Incremental Mark-and-Sweep***. The collection is a standard snapshot-at the-beginning incremental mark-sweep algorithm [19] implemented with a weak tricolor invariant [13]. Metronome extends traversal during marking so that it redirects any pointers pointing at from-space so they point at to-space. Therefore, at the end of a marking phase, the relocated objects of the previous collection can be freed.

Allocation in Metronome is performed using ***segregated free lists***. Memory is divided into fixed-sized pages, and each page is divided into blocks of a particular size. Objects are allocated from the smallest size class that can contain the object.

Since fragmentation is rare, objects are usually not moved. If a page becomes fragmented due to garbage collection, its objects are moved to another (mostly full) page containing objects of the same size [9]. Therefore, Metronome, by nature, is ***mostly non-copying***.

And finally, Relocation of objects is achieved by using a forwarding pointer located in the header of each object [10]. A read barrier maintains a to-space invariant (mutators always see objects in the to-space).

To sum up, Metronome is a hard real-time incremental collector which uses a hybrid of non-copying mark-sweep collection (in the common case) and copying collection (when fragmentation occurs).

4.2 Syncopation

Metronome is a quite good real-time garbage collector that is able to meet the hard constraints of a real-time system. However, it is not good enough. Metronome has some serious issues from a consumer point of view. It is uni-processor collector. It does not have any provision to take advantage of the parallelism that may be present in the computer system. However, maybe we can live with that because after all, it does what it intends to do - performs in real time. However, amongst other issues, another is that the utilization provided by Metronome is not satisfactory enough. The utilization that is offered by Metronome is around 50%. That means, the collector, given that implemented correctly, will guarantee a utilization of 50%. However, this number *50* is probably not good enough.

The creators of Syncopation tried to overcome these short-comings of Metronome. They have done so by undergoing a series of mathematical derivations to determine the utilization u analytically, and thereby noticing that the utilization in a real-time systems depends greatly on the allocation rate. The rate by which allocation happens in heap controls the level of utilization. For conciseness, we do not present those mathematical calculations here. However, noticing that if we can somehow control the allocation rate into the heap, then we can achieve even greater utilization *while not losing the appeal of guaranteeing real-timeliness.*

The solution approach undertaken to make Syncopation is quite interesting. They undertake ***generational*** garbage collection. Now, they do not look at the *nursery* space as we usually do when visualizing a generational garbage collector. They think of the nursery space as a ***filter*** that controls the allocation rate into the *mature* space. This is based on the famous generational hypothesis, that ***most objects die young***. Therefore, nearly at the same ratio of the survival of objects, they find their way into the mature space. Therefore, the overall pressure of allocation into the mature space is quite minimized.

Of course, this introduces new problems to keep the entire setup real-time, after all, that is our core goal. To do so, Syncopation does ***synchronous nursery collection***, along with traditional metronome collection in the mature space. However, that is not their core contribution. Their contribution has been identifying that the allocation rate into the heap is not uniform. If it is uniform, then the filtering nursery space will feed to the mature space in a unform rate as well, and the entire behaviour will also be predictable. But, as the application requirement arises, there can come times when the application rate will experience a spike. Then, the nursery will fill more than once per one real-time interval Δt, and therefore the collector will fall behind the mutator. Thus, the collector will have to run multiple synchronous nursery collections, and the real-time contract may be violated.

Therefore, to meet the irregular much larger allocation rate, Syncopation does one of the two (the choice is parameterized): it either controls ***allocation*** or it controls ***collection***. To stop the nursery from getting too much filled up (and thus stopping multiple synchronous nursery collection that may result in violating the real-time contract), we can control the allocation and not allocate in the nursery at all, directly allocate in the mature space. Therefore, temporarily, the pressure into the mature space will rise from ηa to a, but we can guarantee that we will not have to undergo multiple synchronous nursery collection, and therefore, we can maintain the real-timeliness.

On the other hand, another approach is controlling the collection itself: unconditionally tenuring the remaining unevacuated objects from nursery by logically moving the nursery into the mature space. These operations are made possible through read-barriers.

The naming of Syncopation is also from a musical ideology. In musical terms, syncopation means going off the regular beats. It is a disturbance or interruption of the regular flow of rhythm. Syncopated rhytms are quite funny to listen to, because they break the monotonous beat-pattern and forces the listener to *really listen* (of course, given that the listener is not tone-deaf). Our collector is called ***Syncopation*** because it goes out of its usual way to handle the irregular spike of allocation rate.

Of course, its not that Syncopation will always outperform Metronome. If the survival ratio is almost 1, then the nursery can hardly filter any object out and reduce the allocation pressure into the mature space. In those cases, we are better off using Metronome.

4.3 Shenandoah

The idea behind Shenandoah is to add a forwarding pointer to each object, and requiring all uses of that object to go through the forwarding pointer. Shenandoah is concurrent, and specifically designed for applications where the heap size is very large. In so large heaps, there is no choice but to make the collector concurrent. If the collector is concurrent, it introduces additional bottleneck. The main problem is that in a concurrent collector, moving a potentially in-use object, we also have to atomically update all references to that object to point to the new location. 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.

To achieve this target, Shenandoah divides the entire heap into some regions, each of equal size. 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. Shenandoah uses dome heuristics (mentioned in the previous section) to determine when to run a GC cycle. In a GC cycle, Shenandoah determines the regions that need to be collected, and evacuates live objects from those regions into another region.

Also, it is to be mentioned that Shenandoah is a *regionalized* collector. It is not a generational collector. It uses some heuristics that observe the heap regions and determines which regions are to collect, when to collect. Then, the live objects are moved from the regions to entirely a new region.

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.

Shenandoah does 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.

5 PERFORMANCE ANALYSIS

Gathering the insights of the three collectors, in this section we present them in a categorized and comparative manner to summarize the performances of these collectors under certain conditions. The summary is presented in Table 1.

6 SUGGESTIONS

In this section, we identify some potential area of research for the three collectors.

6.1 Metronome

Metronome is a fairly well optimized hard real-time collector. However, there are lots of research potential here.

6.1.1 Allocation. Allocation scheme used in Metronome is free-list, and therefore will provide poor locality. It may be still acceptable because the intention is to use develop a collector for the real-time systems. So, collection must be done very quickly, which mark-and-sweep allows. Still, this scheme is obviously fragmentation prone and once a while, Metronome has to evacuate to handle the fragmentation. Therefore, there may be scope of work here to organize the heap using bump allocation altogether, and maybe use reference counting for faster collection. Of course, then, allocator will have to participate in handling the counts as well, which may be costly for the real-time system.

6.1.2 Generational. Another obvious area of research is making it generational, which was later done in Syncopation.

Table 1. Comparisons among modern garbage collectors

	Metronome	**Syncopation**	**Shenandoah**
Supported Processors	Uni-processor	Multi-processor	Muti-processor
Real-Time	Yes	Yes	No
Concurrent	No	No	Yes
Genera-tional	No	Yes	No
Forwarding Pointers	Not used	Not used	Used
Supports Large Heap	No	No	Yes
Allocation	Segregated Free List	Segregated Free List	Regionalized Bump Allocation
Collection	Sweep	Nursery to Mature: Evacuation, Mature: Sweep	Evacuation
Triggering a GC Cylcle	Time based	Time based	Heuristics
Copying or not	Mostly non-copying	Mostly non-copying	Fully Copying
Use of Read Barriers	Yes	Yes	Yes
Use of Write Barriers	No	No	Yes
Suitable Application Scenario	Real-time systems with high survival ratio of objects	Real-time systems with low survival ratio of objects	Applications that need very large heap

6.2 Syncopation

Syncopation improves Metronome by regulating the allocation rate in the heap using a nursery space. It has some very interesting points to improve as well.

6.2.1 Allocation. In the mature space, Syncopation still uses Metronome. So, the same allocation improvements can be made here as well.

6.2.2 Controlling Syncopation. In their languages, Syncopation means going off the regular rate of allocation rate into the mature space by controlling either allocation or collection. So, to handle the sudden spikes in the allocation rates, Syncopation moves back and forth between different allocation or collection schemes concerning the nursery. When the allocation rate is not too much, the allocation can be made in nursery. When there is a spike in the rate, Syncopation allocates objects directly in the mature space while allocating. While collecting, it unconditionally moves objects from nursery to mature space.

Instead of doing this back-and-forth alternation of schemes, what we could do is make the nursery itself adaptive. Observing the application behavior, the nursery will be logically sized to meet the pressure of the allocation.

6.2.3 Controlling the Time Δt. Metronome and Syncopation both work using time-based scheduling. The time after which the "tick" will happen is predefined. A predictive model could be trained to predict the optimal Δt for the best real-time performance.

6.2.4 Making it Concurrent. Syncopation is not concurrent. Research efforcts can be made so that the mutator and collectors can run concurrently.

6.3 Shenandoah

Shenandoah is a concurrent garbage collector that lowers the number of pauses and the pause times.

6.3.1 Use of SATB. Shenandoah uses Snapshot at the Beginning (SATB) concurrent marking algorithm. Some sort of incremental algorithm can be used for greater performances.

6.3.2 Making Shenandoah Generational. Shenandoah is rezionalized, it does not exploit the advantages of generational GCs. Although the creators did look into making it generational, the references from old to new space was too cumbersome to work with. However, this is a potential area of research.

6.3.3 Use RC. Sheandoah has been made concurrent (which is its most appealing feature) by working with references. Therefore, employing reference counting can be very promising here.

6.3.4 Heuristics. Shenandoah uses some heuristics to determine when to invoke a GC cycle. The heuristics are very plain and simple, and has lots to improve.

6.3.5 Making it Completely Cncurrent. In a GC cycle of Shenandoah, twice, all the mutator threads are paused for initializing the next operations. Doing these initializations as well concurrently is also a potential area to explore.

7 CONCLUSION

With time, the managed languages with integrated garbage collectors have gained particular popularity. With more reliability in memory management, they have also improved productivity. Moreover, smart and automatic collection has made the performance improve as well.

However, with the outburst of technology, it is also important to note that the variety of applications that need to be accommodated has increased as well. Therefore, it is almost impossible to find a collector that is universally good for all application scenarios. We need to attenuate the collectors for various different application scenarios, and while developing, we need to find the collector that is useful for that particular application.

Of the three collectors we studies, Metronome is useful for real-time applications where the object survival ratio is large. For the applications where this ratio is small, we will find Syncopation more useful. And the applications that require a very large heap size, it is best to use Shenandoah. None of these are free of flaws and have lots to improve.

REFERENCES

Bowen Alpern, C Richard Attanasio, John J Barton, Michael G Burke, Perry Cheng, J-D Choi, Anthony Cocchi, Stephen J Fink, David Grove, Michael Hind, et al. 2000. The Jalapeno virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238.

Joshua S Auerbach, David F Bacon, Florian Bömers, and Perry Cheng. 2007. Real-Time Music synthesis in Java using the Metronome Garbage Collector.. In *ICMC*.

David Bacon et al. [n. d.]. A Mostly Non-Copying Real-Time Collector with Low Overhead and Consistent Utilization. *IBM TJ Watson Research* 42 ([n. d.]).

David F Bacon, Perry Cheng, and David Grove. 2004. Garbage collection for embedded systems. In *Proceedings of the 4th ACM international conference on Embedded software*. ACM, 125–136.

David F Bacon, Perry Cheng, David Grove, and Martin T Vechev. 2005. Syncopation: generational real-time garbage collection in the metronome. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 183–192.

David F Bacon, Perry Cheng, and VT Rajan. 2003a. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 81–92.

David F Bacon, Perry Cheng, and VT Rajan. 2003b. A real-time garbage collector with low overhead and consistent utilization. *ACM SIGPLAN Notices* 38, 1 (2003), 285–298.

Henry G Baker Jr. 1978. List processing in real time on a serial computer. *Commun. ACM* 21, 4 (1978), 280–294.

Daniel G Bobrow and Daniel L Murphy. 1967. Structure of a LISP system using two-level storage. *Commun. ACM* 10, 3 (1967), 155–159.

Perry Cheng and Guy E Blelloch. 2001. A parallel, real-time garbage collector. *ACM SIGPLAN Notices* 36, 5 (2001), 125–136.

David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*. ACM, 37–48.

Christine H Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM, 13.

Richard Jones and Rafael D Lins. 1996. Garbage collection: algorithms for automatic dynamic memory management. (1996).

John McCarthy. 1978. History of LISP. In *History of programming languages I*. ACM, 173–185.

Ernest T Pascarella, Christopher T Pierson, Gregory C Wolniak, and Patrick T Terenzini. 2004. First-generation college students: Additional evidence on college experiences and outcomes. *The Journal of Higher Education* 75, 3 (2004), 249–284.

Generation Scavenging. 1984. A Non-Disruptive High Performance Storage Reclamation Algorithm, by David Ungar, ACM Software Engineering Notes. In *SIGPLAN Notices: Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA*.

Rifat Shahriyar, Stephen M Blackburn, and Daniel Frampton. 2013. Down for the count? Getting reference counting back in the ring. *ACM SIGPLAN Notices* 47, 11 (2013), 73–84.

David Ungar. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM Sigplan notices*, Vol. 19. ACM, 157–167.

Taiichi Yuasa. 1990. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software* 11, 3 (1990), 181–198.