C4: The Continuously Concurrent Compacting Collector

Gil Tene
Azul Systems Inc.
gil@azulsystems.com

Balaji Iyengar
Azul Systems Inc.
balaji@azulsystems.com

Michael Wolf
Azul Systems Inc.
wolf@azulsystems.com

Abstract

C4, the Continuously Concurrent Compacting Collector, an updated generational form of the Pauseless GC Algorithm [7], is introduced and described, along with details of its implementation on modern X86 hardware. It uses a read barrier to support concurrent compaction, concurrent remapping, and concurrent incremental update tracing. C4 differentiates itself from other generational garbage collectors by supporting simultaneous-generational concurrency: the different generations are collected using concurrent (non stop-the-world) mechanisms that can be simultaneously and independently active. C4 is able to continuously perform concurrent young generation collections, even during long periods of concurrent full heap collection, allowing C4 to sustain high allocation rates and maintain the efficiency typical to generational collectors, without sacrificing response times or reverting to stop-the-world operation. Azul systems has been shipping a commercial implementation of the Pauseless GC mechanism, since 2005. Three successive generations of Azul's Vega series systems relied on custom multi-core processors and a custom OS kernel to deliver both the scale and features needed to support Pauseless GC. In 2010, Azul released its first software-only commercial implementation of C4 for modern commodity X86 hardware, using Linux kernel enhancements to support the required feature set. We discuss implementation details of C4 on X86, including the Linux virtual and physical memory management enhancements that were used to support the high rate of virtual memory operations required for sustained pauseless operation. We discuss updates to the collector's management of the heap for efficient generational collection and provide throughput and pause time data while running sustained workloads.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Dynamic storage management; Concurrent programming structures; D.3.4 [*Processors*]: Memory management (garbage collection); D.4.2 [*Storage Management*]: Garbage collection; Virtual memory

General Terms Algorithms, Design, Languages, Performance.

Keywords concurrent, garbage collection, pauseless, generational, read barrier, virtual memory, Linux.

ISMM'11, June 4–5, 2011, San Jose, California, USA.

1. Introduction

Generational collectors are based on the *Weak generational hypothesis* [18] i.e. most objects die young. Therefore by focusing the GC efforts on these objects you would get the proverbial most bang for the buck. Generational focus helps GC algorithms keep up with higher allocation rates, which translates to higher throughput. Most generational collectors divide the heap into two generations, with the younger generation having a smaller, more frequently collected live set. This results in the mutators being exposed to shorter application disruptions associated with young generation collection in the common case, while allowing GC to keep up with high allocation rates. It also serves to delay old generation processing and make it more rare, reducing the frequency of the larger GC workloads and response time artifacts generally incurred in full heap collections.

Most generational collector implementations use stop-the-world, copying, parallel, young generation collectors, relying on the generally short GC pauses of parallel young generation collection to be acceptable for sustained server operations. However, while young generation stop the world pauses are generally quite short, common patterns in enterprise programs executing at current server scales can cause occasional (and sometimes frequent) large young generation GC pauses as large live sets with medium life spans appear in the young generation and are copied within it before being promoted, or as shifts in program execution phases create large amounts of new long-lived objects in rapid succession. The former case is common with in memory object caches, with fat-state session based enterprise applications (such as portals), and with replicated in-memory data and messaging systems. The latter case occurs at application phase shift times, such as during application startup, application failover, cache hydration times, and catalog or contents update or reload operations driven by higher level business processes.

The problems caused by impacts on response time, due to growing heap sizes coupled with stop-the-world-compaction old generation collectors, are well known [3, 9]. However, as Managed Runtimes continue to grow and attempt to use 10s (or even 100s) of Gigabytes of memory the lengths of occasionally long young generation stop-the-world events also grow to become unacceptable for most enterprise applications. When viewed from the perspective of current commodity server environments containing 100s of Gigabytes of cheap RAM, even "modest sized" applications would be expected to sustain live sets in the 10s of Gigabytes, with young generation live sets easily ranging into the Gigabytes and occasionally spiking into the 10s of gigabytes. Concurrent young generation collection in such environments is just as important as concurrent old generation collection was a decade ago, if not more so.

At the core of the problem lie compaction and object relocation: Object relocation is inherent to all effective generational collectors. It would be "extremely hard" to implement a generational collection scheme without supporting promotion. In order for young

generation collection to be concurrent, it must support concurrent object relocation. Concurrent relocating collectors have been proposed in various works [13]. However, with the exception of Azul's JVMs, no commercially shipping concurrent compacting or concurrent relocating collectors are available as of this writing, for either Java or .NET server environments.

C4 is a Generational, Continuously Concurrent Compacting Collector algorithm. It is a throughput-friendly, multi-generational improvement to the full heap, single generation read barrier based Pauseless GC algorithm [7]. All generations in C4 use concurrent compacting collectors and avoid the use of global stop-the-world operations, maintaining concurrent mutator operation throughout all phases of each generation. The cycles and phases of the different generations can be simultaneously and concurrently executed. While current C4 implementations use two generations (young and old), the algorithm can be straightforwardly extended to an N generation system.

C4 is currently shipping as part of commercially available JVM's. These JVMs are available on Azul's custom Vega hardware platform as well as through its recently released X86 based Zing software platform. C4 has also been demonstrated within an OpenJDK based implementation as part of the Managed Runtime Initiative [2]. The C4 collector allows JVMs to smoothly scale up to 10's and 100's of Gigabytes in heap sizes and live object sets, sustain multi-Gigabyte-per-sec allocation rates, and at the same time contain the JVM's measurable jitter[1] and response time artifacts to the low 10's of msec. At the point of this writing, measured JVM jitter is no longer strongly affected by garbage collection, with scheduling and thread contention artifacts dominating.

The X86 based implementation of C4 runs on top of a modified Linux kernel that delivers a new virtual memory subsystem used to support the features and throughput needed by C4's concurrent operation and page lifecycle. The goals of this paper are to describe:

1. The basic C4 algorithm, including inter-generational concurrency and page lifecycle aspects
2. The features added to the Linux virtual memory subsystem to support C4 functionality
3. C4's updated heap management logic

2. The C4 Algorithm

C4 is generational, concurrent, and always-compacting. It uses two independently running instances of a modified Pauseless GC algorithm [7] to simultaneously collect both the young generation and old generation. Each generation's GC cycle goes through logically sequential object Mark, object Relocate and reference Remap phases. The cornerstones of C4's concurrent compacting algorithm include:

2.1 The Loaded Value Barrier (LVB)

The Loaded Value Barrier is an incarnation of Pauseless GC's read barrier. The LVB imposes a set of invariants on every object reference value as it is loaded from memory and made visible to the mutator, regardless of use. The two invariants imposed on all loaded reference values are:

- All visible loaded reference values will be safely "marked through" by the collector, if they haven't been already.
- All visible loaded reference values point to the current location of the safely accessible contents of the target objects they refer to.

An LVB can obviously encounter loaded reference value that do not meet one of these invariants, or both. In all such cases, the LVB will "trigger" and execute collector code to immediately remedy the situation and correct the reference such that it meets the required invariants, before making it visible to subsequent program operations.

LVB differs from a Brooks-style [6] indirection barrier in that, like a Baker-style [4] read barrier, it imposes invariants on references as they are loaded, rather than applying them as they are used. By applying to all loaded references, LVB guarantees no uncorrected references can be propagated by the mutator, facilitating certain single-pass guarantees.

LVB further differs from both Baker-style and Brooks-style collectors in two key ways:

1. LVB simultaneously imposes invariants both on the reference's target address and on the reference's marking state, facilitating both concurrent relocation and precise wavefront tracing in C4 [19] using a single fused barrier.
2. LVB ensures (and requires) that any trigger of either (or both) of its two invariant tests can be immediately and independently repaired by the mutator using Self Healing behavior (see below), leading to efficient and predictable fast path test execution and facilitating C4's concurrent marking, concurrent relocation, and concurrent remapping characteristics.

2.2 Self Healing

LVB is a self-healing barrier. Since the LVB is always executed at reference load time, it has access not only to the reference value being verified, but to the memory address the value was loaded from as well. When an LVB triggers and takes corrective action, modifying a reference to meet the LVB invariants, it will also "heal" the source memory location that the reference was loaded from by (atomically) storing a copy of the reference back to the source location. This allows mutators to immediately self heal the root cause of each LVB trigger as it occurs, avoiding repeated triggers on the same loaded reference, and dramatically reducing the dynamic occurrence of read barrier triggers. Each reference memory storage location will trigger "at most once" (discounting minute levels of atomicity races in the healing process). Since the number of references in the heap is finite, single pass marking and single pass reference remapping are both guaranteed in a straight forward manner.

Self healing is uniquely enabled by the LVB's semantic position in the code stream, immediately following the reference load operation, and preceding all uses or propagation of the loaded reference value. This semantic proximity to the reference load operation grants the LVB access to the loaded reference's source address, which is required in order to perform the healing actions. Through Self Healing, LVB dramatically reduces the dynamic occurrence of read barrier triggering, making LVB significantly more efficient and predictable than both Brooks style and Baker style barriers, as well as other read barriers that will continue to trigger in the hot code path during certain GC phases.

2.3 Reference metadata and the NMT state

Similarly to the single generation Pauseless GC algorithm [7], C4 tracks metadata "Not Marked Through" (NMT) state associated with all object references in the heap[2]. On modern 64 bit hardware, this metadata is tracked in bits of the object reference that are not interpreted as address bits, but are considered by the LVB. Object

[1] JVM jitter, in this context, refers to application observable time delays in executions, that would have been instantaneous had it not been for the behavior of the JVM and the underlying system stack. For example, a 1 msec, Thread.sleep() would be expected to wake up within 1 msec, and any observable delay beyond 1 msec can be attributed to JVM jitter

references with an NMT state that does not match the currently expected NMT value will trigger the LVB.

C4's use of NMT state differs from Pauseless [7]. Where Pauseless maintained a single, global, currently expected value of the NMT state, C4 maintains a different expected value for the NMT field for each generation. In addition, C4 uses reference metadata bits to track the reference's generation (the generation in which the object to which the reference is pointing, resides). This allows the LVB to efficiently verify the NMT value for the proper generation. Since young generation and old generation collections can proceed independent of each other, their expected NMT states will often be different. A reference's generation, however, will never change under C4 without the object it is pointing to being relocated.

If, during an active mark phase, the LVB encounters a loaded object reference value with an NMT state that does not match the current expected state for that reference's target generation, the LVB will correct the situation by changing the NMT state to the expected value, and logging the reference on the collector's work list to make sure that it is safely traversed by the collector. Through self healing, the contents of the memory location that the reference value was loaded from will be corrected as well.

2.4 Page protection and concurrent relocation

C4 uses the same underlying page protection scheme introduced in Pauseless [7]. Pages that are currently being compacted are protected, and the LVB triggers when it encounters a loaded reference value that points to a protected page. In order to correct the triggering situation, the LVB will obtain the new location of the reference's target object, correct the reference value, and heal the contents of the memory location that the reference value was loaded from. In cases where the triggering reference value points to an object that has not yet been relocated, the LVB will first cooperatively relocate the object, and then correct the reference to point to its new location.

2.5 Quick Release

C4 uses the Quick Release method, first introduced in Pauseless [7] and with later variants used in Compressor [13], to efficiently recycle physical memory resources without needing to wait for a GC cycle to complete. When relocating objects for compaction, C4 stores object forwarding information outside of the page that objects are being compacted away from ("from" pages). This forwarding information is later used by the LVB and by the collector's remap phase to correct references to relocated objects such that they point to the object's current address. Once all the objects in a "from" page have been relocated elsewhere, the contents of the "from" page are no longer needed. Quick Release leverages the fact that while the "from" page's virtual address cannot be safely recycled until all live references to objects that were relocated from it are corrected, its physical backing store can be immediately freed and recycled.

C4 uses Quick Release to support hand-over-hand compaction, using the physical memory released by each compacted page as the target resource for compacting the next page. An entire generation can be compacted in a single cycle using a single free seed page, and without requiring additional free memory. As a result, C4 does not need survivor spaces, pre-allocated "to" spaces, or an amount of free memory equal to the size of the live set in order to support single pass compaction.

Quick Release also allows memory to be recycled for mutator allocation needs without waiting for a complete heap remap to complete. This significantly reduces the time between deciding to start a GC cycle and the availability of free memory to satisfy mutator allocation, resulting in GC heuristics that are more simple and robust.

[2] We do not use the tricolor abstraction to represent reference traversal states. Traditional tricolor abstraction [20] applies to object traversal states and not to reference traversal states. "Not Marked Through" references that would trigger an LVB could exist in traditionally white or grey objects and could be referencing objects that are traditionally white, grey or black.

2.6 Collector Phases

Supported by the strong invariants of the LVB, the C4 mechanism is quite straightforward, with each generation's collector proceeding through three phases:

- The Mark Phase: This phase is responsible for tracing the generation's live set by starting from the roots, marking all encountered objects as live, and all encountered object references as marked through. At the beginning of a Mark phase, all live references are known to have an NMT value that matches the previous cycle's expected NMT value. As the Mark phase commences, the collector flips the expected NMT value, instantly making all live references "not marked through", arming the LVB to support single pass marking. The collector proceeds to prime its worklists with roots, and continues to mark until the work lists are exhausted, at which point all reachable objects in the generation are known to have been marked live and all reachable reference NMT states are known to be marked through.
- The Relocate Phase: This phase compacts memory by relocating live objects into contiguously populated target pages, and freeing the resources occupied by potentially sparse source pages. Per-page liveness totals, collected during the previous mark phase, are consulted in order to focus relocation on compacting the sparsest pages first. The core relocation algorithm in C4 is similar to Pauseless. Each "from" page is protected, its objects are relocated to new "to" pages, forwarding information is stored outside the "from" page, and the "from" page's physical memory is immediately recycled.
- The Remap Phase: Lazy remapping occurs as mutator threads encounter stale references to relocated objects during and after the relocate phase. However, in order to complete a GC cycle a remap pass is needed. During the remap phase, all live references in the generation will be traversed, and any stale references to objects that have been relocated will be corrected to point to the current object addresses. At the end of the remap phase, no stale references will exist, and the virtual addresses associated with "from" pages relocated in the relocate phase can be safely recycled. While the remap phase is the third and final logical step of a GC cycle, C4 is not in "a hurry" to finish it. There are not physical resources being held, and lazy remapping can continue to occur without disrupting mutator operations. As a result, C4 fuses each remap phase with the next GC cycle's mark phase for efficiency. Since both phases need to traverse the same object and reference graphs, it joins them into a combined Mark-Remap phase that performs both operations in a single combined pass. Figure 1 depicts, the interleaving of combined Mark-Remap phases across overlapping GC cycles.

3. Multi-Generational Concurrency

By leveraging the generational hypothesis [18], generational collectors tend to derive significant efficiency, and are able to support significantly higher allocation throughput than their single generation counterparts. However, since concurrent relocation of objects seems to be "hard" to do in most currently shipping collectors, their young generation collectors are inherently stop-the-world, and are exposed to significant pause time effects that arise from large transient behaviors or a significant allocation rate of mid and long lived objects.

On the other hand, concurrent single generational collectors, even when they are able to avoid or delay the use of stop-the-world

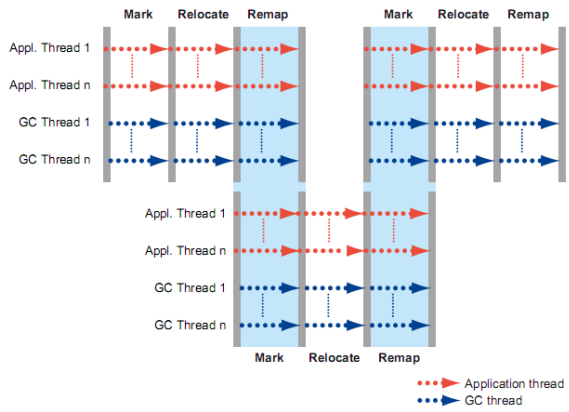

Figure 1. C4 GC cycle

object relocation, are exposed to the inefficiency issues and limited sustainable allocation rates that arise from the lack of a generational filter.

C4 eliminates this classic throughput vs. latency trade-off by supporting concurrent relocation. C4 is simultaneously concurrent in both the young and old generations, and thereby does not suffer from either of the above pathologies. C4 maintains the high throughput, efficiency and sustained allocation rates typical to generational collection while at the same time maintaining the consistent response times and robustness of concurrent compacting collectors.

Both young and old generations use the same concurrent marking, compacting, and remapping algorithm described above. C4 uses a classic card marking mechanism [12] for tracking the remembered set of references from the old generation to the young generation. C4's card marks are precise, supporting a reliably unique card mark per word in memory. C4 uses a filtering Stored Value Barrier (SVB), a write barrier that only marks cards associated with the storing of young generation references into the old generation. The SVB leverages the same generational reference metadata state used by the LVB in order to efficiently filter out unnecessary card marking operations without resorting to address range comparisons and fixed generation address ranges.

3.1 Simultaneous Young and Old generation operation

Most current generational collectors that include some level of concurrent old generation processing interleave stop-the-world young generation collector execution with longer running old generation collection. This interleaving is necessary, as without it the sustainable allocation rate during old generation collection would be equivalent to that of a non-generational collector, negating the value of having a young generation collector to begin with. Simultaneous collection of both generations is not commonly supported. Such simultaneous operation would not add value to already stop-the-world young generation collectors. When both the old and young generations use concurrent collectors, allowing concurrent young generation collection to execute during a concurrent old generation collection is similarly necessary. Straight forward interleaving may suffice for maintaining throughput. For example, the old generation collector can be "paused" while the young generation collector executes a complete cycle. However, such interleaving would limit flexibility and increase code complexity by forcing the inclusion of relatively fine grain synchronization and "safe pausing" points in the old generation collector's mechanism, and potentially in the young generation collector's as well.

C4 supports simultaneous concurrent execution of both the old and young generation collectors, limiting cross-generational synchronization to interlocks that are used to synchronize some phase shift edges as well as access to critical data structures. Figure 2 illustrates this.

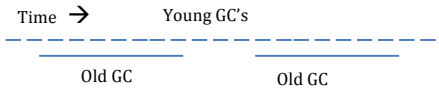

Figure 2. Simultaneous generational collection in C4

In order to facilitate continued execution of multiple young generation cycles during an old generation mark phase we avoid performing a full heap collection that would prohibit changes to the young generation's expected NMT state. Instead, we perform old generation-only marking using a set of young-to-old roots generated by a special young generation cycle that is triggered along with every old generation mark cycle. This young generation cycle hands off the young-to-old pointers it locates to the old collector for use as part of its root-set. The hand-off is done during the marking phase of the young generation cycle with the old generation collector concurrently consuming the root set produced by the young generation marker. Further young generation cycles continue to happen concurrently, and will ignore the new-to-old pointers they find during marking. While the young generation collector is mostly decoupled from the old generation collector, synchronization points around critical phase shifts are necessary. For example, the young collector does need to access objects in the old generation. If a card mark is set for a pointer in an object in the old generation, the young generation collector must read that pointer to discover which object it refers to. Similarly, Klass objects in the old generation may describe the structure of the objects in young generation memory. For the young generation collector to find the pointers in an object it needs to mark through, it must read that object's corresponding klass object to determine the layout of the object at hand. In either case, since the old collector might be in the midst of relocating the object that is of interest to the young generation, some level of synchronization around cross-generational access is necessary.

We resolve these and other issues by introducing a synchronization mechanism between the two collectors referred to as 'interlocks'. An interlock between the old and new collector, briefly halts one of the collectors, to provide safe access to the object at hand. The number of Klass objects is typically a tiny fraction of the objects in memory, and this results in a very short serialization between the young and the old generation cycles, while the mutator maintains concurrency. We similarly use the interlock mechanism to allow the young generation collector to safely scan card marks. The young generation collector waits until all old generation relocation is halted, and then performs the card mark scan. Once done, it signals the old collector that relocation may resume. The synchronization is done at page granularity, so the affected collector is delayed only for short periods of time, or the order of what it takes for the old collector to relocate the live objects in a single page.

The two collectors need to synchronize at other points as well. E.g. each collector needs exclusive access to VM internal data structures such as the system dictionary, the symbol table and the string tables. The batched memory operations described in section 5 also require the collectors to coordinate batch boundaries between them.

4. Implementation notes

The C4 *algorithm* is entirely concurrent, i.e. no global safepoints are required. We also differentiate between the notion of a global safepoint, where are all the mutator threads are stopped, and a checkpoint, where individual threads pass through a barrier function. Checkpoints have a much lower impact on application responsiveness for obvious reasons. Pizlo et al [16] also uses a similar mechanism that they refer to as ragged safepoints.

The current C4 *implementation*, however, does include some global safepoints at phase change locations. The amount of work done in these safepoints is generally independent of the size of the heap, the rate of allocation, and various other key metrics, and on modern hardware these GC phase change safepoints have already been engineered down to sub-millisecond levels. At this point, application observable jitter and responsiveness artifacts are dominated by much larger contributors, such as CPU scheduling and thread contention. The engineering efforts involved in further reducing or eliminating GC safepoint impacts will likely produce little or no observable result.

Azul has created commercial implementations of the C4 algorithm on three successive generations of its custom Vega hardware (custom processor instruction set, chip, system, and OS), as well on modern X86 hardware. While the algorithmic details are virtually identical between the platforms, the implementation of the LVB semantics varies significantly due to differences in available instruction sets, CPU features, and OS support capabilities.

Vega systems include an LVB instruction that efficiently implements the entire set of LVB invariant checks in a single cycle, and is assisted by a custom TLB mode that supports GC protected pages and fast user mode traps. The custom OS kernel in Vega systems includes support for extremely efficient virtual and physical memory management and manipulation, facilitating efficient page protection and direct support for quick release and C4's overall page lifecycle needs.

Azul's LVB implementation on modern X86 hardware maintains the same semantic set of operations and invariant checks using a set of X86 instructions, effectively "micro-coding" the LVB effect and interleaving it with other instructions in the X86 pipeline. The actual implementation of an LVB sequence varies significantly even between different parts of a single runtime. For example, LVBs appear in interpreter code, in JIT-compiled code (coming from two different levels of tiered, optimizing JIT compilers), and in a multitude of places in the C++ implementation of runtime code. While each has different implementations of slow paths, fast paths, instruction scheduling and interleaving opportunities, the same LVB semantics and invariants are maintained in all cases. For the purposes of this paper, the abstraction presented earlier and proves to be sufficient, as a full description of the various LVB implementation options and details warrants a separate research paper. However, see Appendix A for a sample implementation.

4.1 Page life cycle

Unlike most existing collectors which tend to use relatively static memory mappings, the C4 algorithm uses a dynamic page life cycle that includes continuous mapping, remapping, protection, and unmapping of memory pages as GC cycles proceed. Figure 3 depicts the various states in the life cycle of a young generation heap page (old generation heap pages go through a similar life cycle). The solid rectangles represent virtual pages with backing physical storage. The dashed rectangles represent virtual pages with no backing physical storage. The solid oval represents a physical page that hasn't been mapped to a virtual page yet. The page state is represented as a tuple: <State><Gen><Prot>.

These states map to C4's phases. Active pages start their lifecycle in the *Allocating* state, representing a virtual memory page mapped to a physical backing store, into which allocated objects are placed. *Allocating* pages transition to the *Allocated* state once their memory space is filled up with allocated objects. Pages remain in the *Allocated* state until the next relocation phase, when C4 chooses to compact pages below a certain liveness threshold. A page selected for compaction transitions to the *Relocating* state and is protected from mutator access. Each live object in a *Relocating* page is moved to a new, compacted page either by the first mutator thread to access it, or by the collector. As described in section 2.5, forwarding information that tracks new object locations is kept outside of the *Relocating* page. Once the page contents had been copied out, and the page transitions to the *Relocated* state, its physical memory can be freed. At this point, the virtual page remains in the *Relocated* state, but the physical page it was mapped to is freed and transitions to the *Free* state. We refer to this transition as Quick-Release, where physical resources are recycled well before address space can be reused. The virtual page remains in the *Relocated* state until the end of the next remap phase, at which point all the references pointing to that page would have been remapped to its new location. At that point the virtual page transitions to the *Free* state, from which it can be recycled into the *Allocating* state by mapping it to an available physical memory page.

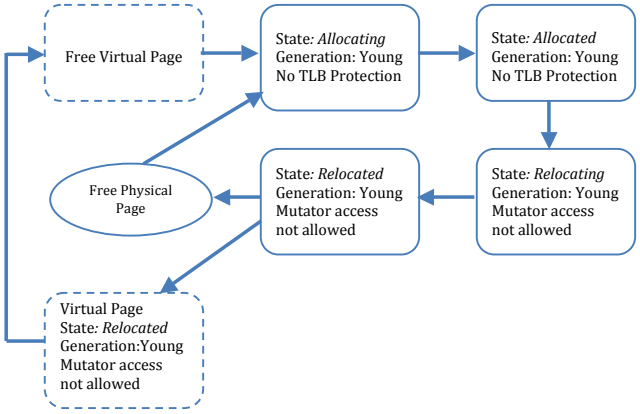

Figure 3. Life cycle of heap page

5. Operating System Support

During normal sustained operation, the C4 algorithm makes massive and rapid changes to virtual memory mappings. In a sequence that was first described in Pauseless [7], page mappings are manipulated at a rate that is directly proportional to the object allocation rate in the application. Allocation results in mapping in a new page, the page is then remapped, protected and unmapped during the course of a GC cycle, resulting in 3 or 4 different mapping changes per page-sized unit of allocation. Later works, such as Compressor [13], share this relationship between sustained allocation rates and virtual memory mapping manipulation rate. The operating system's ability to sustain virtual mapping manipulation rates directly affects the sustainable application throughput that such collectors can maintain without imposing observable application pauses.

In this section, we focus on two key metrics that are critical to sustaining continuous concurrent compaction in C4. We explain the limitations of current Linux virtual memory semantics and their performance implications, and demonstrate how virtual memory subsystem improvements achieved through new APIs and looser semantic requirements deliver several orders of magnitude in improvement for these key metrics.

Active Threads	Linux	Modified Linux	Speedup
1	3.04 GB/sec	6.50 TB/sec	>2,000x
2	1.82 GB/sec	6.09 TB/sec	>3,000x
4	1.19 GB/sec	6.08 TB/sec	>5,000x
8	897.65 MB/sec	6.29 TB/sec	>7,000x
12	736.65 MB/sec	6.39 TB/sec	>8,000x

Table 1. Comparison of sustainable mremap rates

It is typically desirable (for both stability and headroom reasons) for the GC cycles to run at a 1:5 or lower duty cycle. Furthermore, within each GC cycle, page remapping typically constitutes less than 5% of the total elapsed cycle time, with most of the cycle time consumed by marking/remapping and relocation. Together, these ratios qualities mean that the garbage collector needs to sustain a page remapping at a rate that is 100x as high as the sustained object allocation rate for comfortable operation. This places a significant stress on the virtual memory subsystem in stock Linux implementations, and would significantly limit the sustainable allocation rate on C4 unless the OS were able to support the required performance.

5.1 Supporting a high sustainable remap rate

On stock Linux, the only supported memory remapping mechanism has three key technical limitations:

- Each page remap includes an implicit TLB invalidate operation. Since TLB invalidates require multiple cross-CPU interrupts, the cost of remapping grows with the number of active CPU cores in the executing program. This reduction in remap performance with increased thread counts happens even when the active threads do not participate in the remapping, or have any interaction with the remapped memory.
- Only small (4KB on X86-64) page mappings can be remapped.
- Remap operations are single threaded within a process (grabbing a common write lock in the kernel).

To address the main remapping limitations in stock Linux and to support C4's need for sustained remapping rates, we created a new virtual memory subsystem that exposes new APIs and added features that safely support memory remaps, unmaps, and protection changes without requiring TLB invalidation (TLB invalidation can be applied at the end of a large set of remaps if needed). C4 uses these new APIs to manage page life cycles. Our new virtual memory subsystem also supports explicit and mixed mapping and remapping of large (2MB on X86-64) pages, and safely allows concurrent memory manipulation within the same process.

Table 1 has a sample comparison of sustainable remap rates between a stock, un-enhanced Linux and a Linux kernel containing our changes.

- The tests were done on a 16 core system, 4 Socket AMD Barcelona based system with 128GB of memory. The Linux kernel revision is 2.6.32.
- The tests allocate 2MB of memory, and then repeatedly remap it around a large address space for a given duration of time.
- The Active threads represent the number of additional active threads (not the remapping thread) in the test application. The active threads perform a pure integer loop workload - their sole purpose is to "exist" and allow measurement of the effect of running application threads on remap throughput (as a result of needing to send TLB invalidates to additional active CPUs).
- Stock Linux test maps the memory using mmap(), and remaps the memory using a 2MB mremap() call.
- The new virtual memory subsystem maps the memory using az_mmap() [with a flag indicating large pages are to be used], and remaps the memory using a single thread performing 2MB az_mremap() calls [with a NO_TLB_INVALIDATE flag].
- These results do not take into account the additional speedup that can be gained by using multiple remapping threads in our new virtual memory subsystem.

5.2 Supporting a high remap commit rate

For efficient implementation of large amounts of relocation, the current C4 *implementation* places an atomicity requirement on the memory remapping changes in a given relocation phase. Page protection and remapping changes need to become visible to all running application threads at the same time, and at a safe point in their execution. In order to perform a relocation of a bulk set of pages, the C4 Collector will typically bring all application threads to a common safe point, apply a massive remapping of a significant portion of the garbage collected heap, and then allow the threads to continue their normal execution. Since application code execution in all threads is stopped during this operation, a noticeable application pause would result if it cannot be applied in an extremely fast manner. The only operation supported on Stock Linux that can be used to facilitate the C4 remap commit operation is a normal remap (measured in 1). C4 would need to bring all threads to a safe point, and hold them there for the duration of time that it takes Linux to perform the remapping of the bulk page set. To address the Remap Commit Rate limitations in stock Linux, we support a batch preparation of virtual memory operations. Batch operations are performed on a shadow page table without becoming visible to the process threads, and a large batch of remap operations can be prepared without requiring any application threads to be stopped during the preparation phase. The prepared batch is then committed by the GC mechanism using a single "batch commit" operation call to the kernel virtual memory subsystem. Since a batch commit will typically be executed during a global safe point, the batch commit operation is designed to complete in a matter of microseconds, such that no perceivable application pauses will occur.

Table 2 compares stock Linux Remap Commit Rates established in previously described sustainable remap rate tests and the time it would take to perform 16GB of remaps, with the measured rate and time it takes Linux enhanced with our virtual memory subsystem to commit a 16GB batch of remap operations (prepared using az_mbatch_remap() calls) using an az_mbatch_commit() call.

Active Threads	Linux	Modified Linux	Speedup
0	43.58 GB/sec (360ms)	4734.85 TB/sec (3us)	>100,000x
1	3.04 GB/sec (5s)	1488.10 TB/sec (11us)	>480,000x
2	1.82 GB/sec (8s)	1166.04 TB/sec (14us)	>640,000x
4	1.19 GB/sec (13s)	913.74 TB/sec (18us)	>750,000x
8	897.65 MB/sec (18s)	801.28 TB/sec (20us)	>890,000x
12	736.65 MB/sec (21s)	740.52 TB/sec (22us)	>1,000,000x

Table 2. Comparison of peak mremap rates

- An Active Thread count of 0 was also included, since commits will happen during a global safepoint, and it is therefore possible that no active threads would execute during the commit operation. However, it should be noted that the number of threads that may be active during the safepoint operations can also be non-zero (which is why multiple thread counts are also modeled in the test). While Application execution is held at a safepoint during the remap commit operation, application threads may still be actively executing runtime code while logically held at the safepoint. E.g. runtime operations that occur under a single bytecode and release the safepoint lock, such as I/O system calls, long runtime calls, and JNI calls, are executed under a

logical safepoint, and may keep cpu cores busy during a remap commit.
- An az_mbatch_commit() call produces a single global TLB invalidate per call.

6. Heap Management

6.1 Tiered Allocation Spaces

C4 manages the heap in 2M sized physical pages. The allocation path uses a Thread Local Allocation Buffer (TLAB) mechanism [10] found in most enterprise virtual machines. Each Java thread uses bump pointer allocation in its local allocation buffer. In supporting concurrent compaction and remapping, C4 must also adhere to a requirement for objects to not span relocation page boundaries. These allocation and relocation schemes, while being fast and supporting consistent application response times, can result in some headroom in the local buffer being wasted. The worst case headroom wastage can be as high as 50%, if a program serially allocates objects of size (N/2+1)MB's or N+1, where N is the larger of the buffer size or the relocation page size. Other work such as [16], try to deal with this waste by allocating fragmented objects while [5] addresses this issue by maintaining two levels of allocation bins. To cap the worst case physical memory waste, while at the same time containing the worst case blocking time of a mutator waiting for a single object to be copied, we bin the objects into three different size ranges, and handle the memory for each size range differently. The three tiers are:

- Small Object Space: Contains objects less than 256 KB in size. The region is managed as an array of 2 MB pages. Objects in this space must not span page boundaries, and new object allocation usually uses TLABs.
- Medium Object Space: Contains objects that are 256 KB and larger, up to a maximum of 16 MB. The region is managed as an array of 32 MB virtual blocks, physically allocated in 2MB units. Objects in this space may span 2 MB pages, but must not cross virtual block boundaries. Objects allocated within a block will be aligned to 4 KB boundaries. We explain the need for this in section 6.3.
- Large Object Space: Contains objects larger than those that would fit in Small or Medium Space (larger than 16 MB). Unlike Small Space and Medium Space where virtual memory is organized into fixed sized chunks, virtual and physical memory is allocated in multiples of 2MB to fit the size of the object being allocated. All objects in this space are allocated on 2MB aligned boundaries, with no two objects sharing a common 2MB page.

The Small and Medium tiers are managed using fixed sized virtual blocks of memory, eliminating the possibility of fragmentation in their memory space. In contrast, the Large Object space does incur virtual address fragmentation. Since Large Space objects all reside in dedicated 2MB pages, compacting the virtual space is a simple matter of remapping the pages containing the objects to new virtual locations using the same relocation scheme described in section 2. As described, the tiered allocation spaces scheme limits the worst case headroom waste to 12.5%. The worst case Small Object Space waste is 12.5% (256KB out of 2MB). The worst case space waste for both the Medium and Large Object Space is 11.1% (2MB out of 18MB). However, arbitrarily small worst case sizes can be similarly engineered by modifying the object size thresholds and block sizes in the scheme, as well as by adding additional tiers.

6.2 Allocation pathway

Objects that fit in the Small Object Space are allocated using TLABs [10], on which we piggy-back the test for object size and fit into the Small Object space. Each TLAB tracks a TLAB end-of-region pointer, and a fast-allocation end-of-region pointer. The TLAB end-of-region indicates the end of the 2 MB TLAB. The fast-allocation end-of-region is checked by the fast path assembly allocation code. It would never point more than 256 KB past the top pointer, thus forcing middle size and large size object allocation into the slow path. The slow path bumps forward the fast-allocation end pointer as needed, until the actual TLAB end-of-region pointer is reached.

Allocation in the Medium Object Space shares allocation blocks across Java threads. A Java thread trying to allocate into a Medium Object Space block claims the required sized chunk by atomically updating the top pointer of the page.

Allocation in the Large Object Space is achieved by atomically updating a global top pointer for the space.

6.3 Relocating medium space objects

Middle space blocks are compacted using an algorithm similar to that of small space pages, combined with support for incrementally copying object contents in 4 KB chunks as they are concurrently accessed. This reduces the copy delays imposed when a mutator tries to access an unrelocated object of non trivial size. Compaction is done with a protected page shattering virtual memory remap operation, followed later by a copying page healing merge operation, defined below:

- Page Shattering Remap operation: Remaps virtual memory in multiples of 4 KB pages to a new virtual address, shattering contiguous 2MB mappings in the process. The new address is protected against both read and write access, forcing protection faults if the contents are accessed prior to page healing.
- Page Heal/Merge operation: A 2 MB aligned compacted region comprised of previously shattered 4KB virtual memory pages is "healed" by the collector to form a 2MB virtual mapping. The healing operation uses a contiguous 2 MB physical page as a copy-on-demand target, sequentially copying the individual 4KB contents to their proper position in the target physical page while at the same time supporting asynchronous on-demand fault driven copying of 4KB section as the mutators access them. Once the 4KB sections of a 2MB aligned region have been copied to the target physical page, their original physical pages become free and are recycled, facilitating hand-over-hand compaction of the Medium Space.

The shattering remap and healing merge operations are among the features in the new Linux virtual memory subsystem discussed in the section 5 . With these two new calls, the relocation algorithm for Medium Space objects becomes:

1. Remap each live object to a new 4KB aligned location in target to-blocks. The new virtual address is shattered and protected in the same operation.
2. Publish the new virtual address as the forwarding pointer for each object.
3. Heal/Merge each 2MB page in the to-blocks in a hand-over-hand manner, using the physical memory released from each 2MB Heal/Merge operation as a target resource for the next Heal/Merge operation.
4. As Mutators concurrently access Middle Space objects during the relocation, faults are used to heal only the 4KB section in which the fault occurred, minimizing the mutator blocking time.

The 4 KB remap granularity is the source of needing to allocate medium sized objects at a 4 KB alignment.

7. Experiments

Our experiments are intended to highlight the behavior of simultaneous generational concurrency by comparing the response time behavior of C4 to that of other collectors. We compared C4 with an intentionally crippled version of C4, with simultaneous generational concurrency disabled (dubbed C4-NoConcYoung), as well as with the OpenJDK HotSpot CMS and Parallel collectors.

C4 is intended to maintain consistent response times in transaction oriented programs running business logic and interactive server logic on modern servers, using multi-GB heaps and live sets, sustaining multi-GB/sec allocation rates, as well as common application and data patterns found in such enterprise applications. With low end commodity servers reaching 96GB or more in main memory capacity in 2011, it is surprisingly hard to find widely used server scale benchmarks that measure the response time envelope of applications that would attempt to use such capacities within managed runtimes. Large, mutating object caches that occupy significant portions of the heap, fat state-full sessions, rapid messaging with in-memory replication, and large phase shifts in program data (such as catalog updates or cache hydration) are all common scenarios in server applications, but benchmarks that require more than 1-2GB of live state per runtime instance (1% of a modern commodity server's capacity) are virtually non-existent. Furthermore, industry benchmarks that do include response time criteria typically use pass/fail criteria that would be completely unacceptable for business applications - with most including average and 90%'ile response times and standard deviation requirements, but no max time, 99.9%'ile, or even 99%'ile requirements.

Since response time behavior under sustainable load is the critical measure of a collector's operational envelope that we are interested in, we constructed a simple test bench that demonstrated the worst case response time behavior of various collector configurations. The test bench measured response times for an identical working set workload run on various collector and heap size configurations. A modified version of the commonly used SPECjbb2005 transactional throughput workload was used, changed to include a modest 2GB live set cache of variable sized objects that churns at a slight newly cached object rate of about 20MB/sec. The test bench also included a mechanism to measure the response time characteristics of transactions throughout the run. The modified SPECjbb2005 workload was run for prolonged constant-load runs with 4 warehouses, and we focused our measurements on worst case response times seen in sustained execution. The tests were all executed on the same hardware setup; a 2 socket, 12 core Intel x5680 server with 96GB of memory. On this platform, the test bench exhibits an allocation rate of about 1.2GB/sec, and live sets were consistently measured at about 2.5GB. All runs were executed at least long enough to proceed through multiple full heap collection cycles, and to allow the normal object churn and heap fragmentation to produce at least one major compaction event. The worst case response time for each run was collected and plotted. Figure 4 shows the results, depicting worst case response time of the various collectors at different heap sizes. For reference, Figure 5 provides the test bench throughput measured for each test[3].

As expected the worst case response times of collectors that do not perform concurrent compaction [i.e. CMS and ParallelGC] start off at multi-second levels, and get significantly worse as the heap size grows, even with the relatively modest 2.5GB live sets maintained across all tests. C4-NoConcYoung, due to its lack of simultaneous generational concurrency, exhibits problems maintaining pauseless operation and low response times when the heap size is not large enough to absorb the 1.2GB/sec allocation rate during full heap collections, and its results improve with heap size (reaching pauseless operation at 24GB). C4 exhibits a wide operating range with consistent response times, and is able to sustain pauseless operation at the test bench's allocation rate even with relatively low amounts of empty heap.

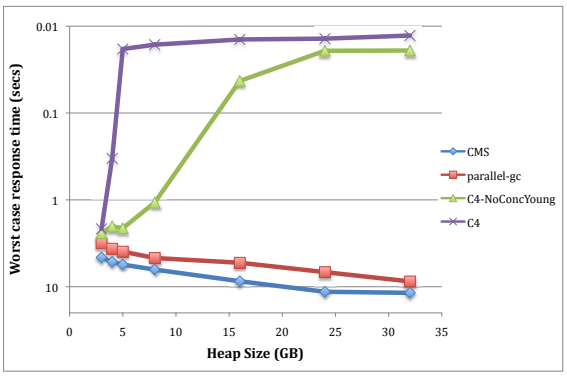

Figure 4. Worst case response times

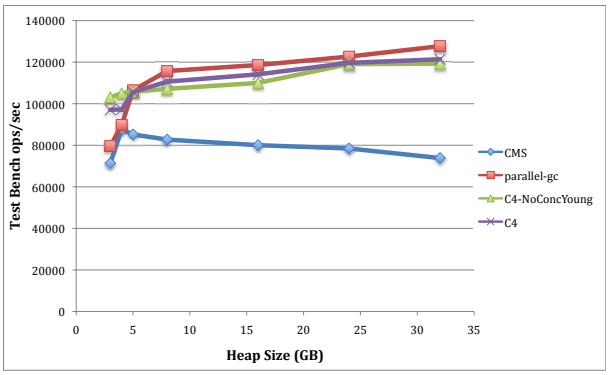

Figure 5. Test Bench Raw Throughput

8. Related Work

The idea of using common page-protection hardware to support GC has been around for a while [1]. Appel et al [1] protect pages that may contain objects with non-forwarded pointers (initially all pages). Accessing a protected page causes an OS trap which the GC handles by forwarding all pointers on that page, then clearing the page protection. Compressor [13] uses techniques similar to our work and Pauseless GC [7], in that they protect the to-virtual-space. They update the roots to point to their referents' new locations at a safepoint and the mutator threads are then allowed to run. Their mutator threads encounter and use access violation traps to fixup the references they follow.

Incremental and low-pause-time collectors are also becoming popular again. Pizlo et al [15] discuss three such algorithms. The Chicken algorithm uses a Brooks-style read barrier and a wait-free "aborting" write barrier that in the fast path only requires a read operation and a branch followed by the original write operation. The Stopless algorithm [14] uses an expanded object model with tem-

[3] While measuring raw throughput is not the objective of these tests (non-SLA-sustaining throughput numbers are generally meaningless for production server environments), Figure 5 shows that C4 closely matches ParallelGC in throughput (to within 1-6% depending on heap size) on this test bench workload. Average latencies, which are also not the objective here, can be directly derived from throughput, knowing that exactly 4 concurrent threads were transacting at all times.

porary object while the Clover algorithm relies on a probabilistic model and writes a random value to a field and does field by field copy. It uses this technique in conjunction with a barrier to provide concurrency.

Attempts to deal with the stop-the-world atomic nature of young generation collection are relatively new. While we are not aware of concurrent young generation collectors, [11] presents a stop-the-world incremental young generation collector for real time systems. The algorithm uses a combination of a read-barrier and a write-barrier to capture remembered set changes during incremental collection.

9. Conclusions

C4 is a generational, continuously concurrent compacting collector algorithm, it expands on the Pauseless GC algorithm [7] by including a generational form of a self healing Loaded Value Barrier (LVB), supporting simultaneous concurrent generational collection, as well as an enhanced heap management scheme that reduces worst case space waste across the board.

C4 is currently included in commercially shipping JVMs delivered as a pure software platform on commodity X86 hardware, and demonstrates a two-orders-of-magnitude improvement in sustainable [worst case] response times compared to both stop-the-world-ParallelGC and mostly-concurrent CMS [8] collectors executing on identical hardware, across a wide range of heap sizes.

The C4 implementation currently supports heap sizes up to 670GB on X86-64 hardware, and while this represents some significant headroom compared to the commonly sold and used servers, commercially available commodity servers holding 1TB and 2TB can already be purchased at surprisingly low cost points as of this writing. As explained in the implementation notes (see Section 4), while the C4 algorithm is fully concurrent (with no required global stop-the-world pauses), the current C4 implementation does perform some very short stop-the-world phase synchronization operations for practical engineering complexity reasons. While these operations now measure in the sub-millisecond range on modern X86 servers, future work may include further improvement, potentially fully implementing the complete concurrent algorithm, and expanding the supported heap sizes to 2TB and beyond.

A. Appendix: Metadata encoding and LVB pseudocode

Section 2 describes C4's Loaded Value Barrier (LVB), its interactions with reference metadata, NMT state, and page protection. The specified LVB behavior and reference metadata encoding can be implemented in a wide variety of ways. This appendix discusses some of the encoding options and presents an example encoding and matching pseudocode for LVB logic.

For LVB to correctly impose the marked-through invariant on loaded references described in section 2.1, metadata describing the reference's NMT state and the reference's generation (the GC generation in which the object that the reference points to resides) must be associated with each reference. The most straightforward way to associate metadata with the reference is to store it within a 64 bit reference field, using bits that are not interpreted as addresses. Ignoring the metadata bits for addressing purposes can be achieved by stripping them off of the reference value ahead of the using the value for addressing purposes. Alternatively, on architectures where such stripping imposes a high overhead, virtual memory multi-mapping or aliasing can be used such that, otherwise identical addresses (for all metadata bit value combinations) are mapped to the same physical memory contents. Our enhanced Linux virtual memory subsystem includes support for efficiently aliasing 1GB aligned address regions to each other, such that any virtual memory operation applied to an address in an aliased region would apply identically to all the addresses aliased to it as well.

The reference metadata bit-field itself can be encoded in various ways. The simplest examples would encode NMT state using a single bit, and encode the reference generation (young or old) using a single, separate bit. However, other considerations and potential uses for metadata encodings exist. For example, for various practical runtime implementation considerations, it is useful to have NULL values and non-heap pointers encoded such that their interpretation as a reference value would not appear to be in either the old or young generations, leading to a useful encoding of a multi-bit "SpaceID" where the young and old generations occupy two of the available non-zero spaceIDs. Additional SpaceIDs can be useful for purposes that are either orthogonal to or outside the scope of C4 (e.g., stack-local and frame local allocation space identifiers, such as those described in [17]). When encodings with a larger number of spaceIDs is desirable, a combined SpaceID+NMT encoding in a common bit field becomes useful for address space efficiency purposes, avoiding the "waste" of an entire bit on NMT state for SpaceIDs that do not require it.

In the interest of simplicity, we describe a simple encoding using 2 bits for SpaceID, and a single bit for recording NMT state. Figure 6 shows the layout of this metadata in a 64 bit object reference. Figure 7 describes the SpaceIDs field value interpretation, and gives pseudocode for an LVB implementation with the metadata encoding that matches the discussion in sections 2.1 through 2.4 .

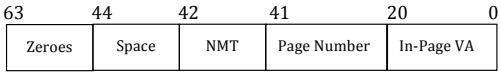

Figure 6. Object Reference Word Layout

References

[1] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, PLDI '88, pages 11–20, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.53992. URL `http://doi.acm.org/10.1145/53990.53992`.

[2] Azul Systems Inc. Managed Runtime Initiative. `http://www.managedruntime.org/`, 2010.

[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. ISBN 1-58113-628-5. doi: http://doi.acm.org/10.1145/604131.604155. URL `http://doi.acm.org/10.1145/604131.604155`.

[4] H. G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21:280–294, April 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359460.359470. URL `http://doi.acm.org/10.1145/359460.359470`.

[5] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: http://doi.acm.org/10.1145/1375581.1375586. URL `http://doi.acm.org/10.1145/1375581.1375586`.

```
struct Reference
{
  unsigned inPageVA  : 21;  // bits 0-20
  unsigned PageNumber: 21;  // bits 21-41
  unsigned NMT       : 1;   // bit 42
  unsigned SpaceID   : 2;   // bits 43-44
  unsigned unused    : 19;  // bits 45-63
};

int Expected_NMT_Value[4] = {0, 0, 0, 0};

// Space ID values:
// 00   NULL and non-heap pointers
// 01   Old Generation references
// 10   New Generation references
// 11   Unused

LVB pseudocode:

doLVB(void *address, struct Reference &value)
{
  int trigger = 0;
  if (value.NMT != Expected_NMT_Value[value.SpaceID])
    trigger |= NMT_Trigger;
  if (value.pageNumber is protected)
    trigger |= Reloc_Trigger;
  if (trigger != 0)
    value = doLVBSlowPathHandling(address, value, trigger);
}

doLVBSlowPathHandling(void* address, struct Reference &value, int trigger)
{
  struct Reference oldValue = value;

  // Fix the trigger condition(s):
  if (trigger | NMT_Trigger) {
    value.NMT = !value.NMT;
    QueueReferenceToMarker(value);
  }
  if (trigger | Reloc_Trigger) {
    if (ObjectIsNotYetRelocated(value)) {
      relocateObjectAt(value);
    }
    value = LookupNewObjectLocation(value);
  }

  // Heal source address:
  AtomicCompareAndSwap(address, oldValue, value);

  return value;
}
```

Figure 7. LVB Pseudocode

[6] 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. ISBN 0-89791-142-3. doi: http://doi.acm.org/10.1145/800055.802042. URL `http://doi.acm.org/10.1145/800055.802042`.

[7] 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. ISBN 1-59593-047-7. doi: http://doi.acm.org/10.1145/1064979.1064988. URL `http://doi.acm.org/10.1145/1064979.1064988`.

[8] D. Detlefs and T. Printezis. A Generational Mostly-concurrent Garbage Collector. Technical report, Mountain View, CA, USA, 2000.

[9] 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. ISBN 1-58113-945-4. doi: http://doi.acm.org/10.1145/1029873.1029879. URL `http://doi.acm.org/10.1145/1029873.1029879`.

[10] D. Dice, A. Garthwaite, and D. White. Supporting per-processor local-allocation buffers using multi-processor restartable critical sections. Technical report, Mountain View, CA, USA, 2004.

[11] D. Frampton, D. F. Bacon, P. Cheng, and D. Grove. Generational Real-Time Garbage Collection: A Three-Part Invention for Young Objects. ECOOP '07, pages 101–125.

[12] U. Hölzle. A Fast Write Barrier for Generational Garbage Collectors. In *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, 1993.

[13] H. Kermany and E. Petrank. The Compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1134023. URL `http://doi.acm.org/10.1145/1133981.1134023`.

[14] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 159–172, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-893-0. doi: http://doi.acm.org/10.1145/1296907.1296927. URL `http://doi.acm.org/10.1145/1296907.1296927`.

[15] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '08, pages 33–44, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: http://doi.acm.org/10.1145/1375581.1375587. URL `http://doi.acm.org/10.1145/1375581.1375587`.

[16] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '10, pages 146–159, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: http://doi.acm.org/10.1145/1806596.1806615. URL `http://doi.acm.org/10.1145/1806596.1806615`.

[17] G. Tene, C. Click, M. Wolf, and I. Posva. Memory Management, 2006. US Patent 7,117,318.

[18] D. Ungar. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM. ISBN 0-89791-131-8. doi: http://doi.acm.org/10.1145/800020.808261. URL `http://doi.acm.org/10.1145/800020.808261`.

[19] M. T. Vechev, E. Yahav, and D. F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '06, pages 341–353, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1134022. URL `http://doi.acm.org/10.1145/1133981.1134022`.

[20] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, 1992. Springer-Verlag. ISBN 3-540-55940-X. URL `http://portal.acm.org/citation.cfm?id=645648.664824`.