Parallel garbage collection for SBCL

Hayley Patton
hayley@applied-langua.ge

ABSTRACT

We describe a parallel garbage collector which we are implementing for Steel Bank Common Lisp. The collector reclaims memory and allows for bump allocation without the collector needing to move objects, using a mark-region heap based on Immix [8]. The heap is comprised of pages, and pages are comprised of lines. We exploit the design of Immix in two ways: (i) generations are implemented without the collector moving objects or recording the generation in each object, by associating generations with lines; and (ii) conservative root finding is implemented by updating an object map only on demand, based on recording runs of contiguously allocated objects. The parallel garbage collector using one core usually is slower than the copying collector of SBCL, outperforms copying with two cores, and continues to scale with more cores.

CCS CONCEPTS

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

ACM Reference Format:
Hayley Patton. 2023. Parallel garbage collection for SBCL. In *Proceedings of the 16th European Lisp Symposium (ELS'23)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.7816398

1 INTRODUCTION

Steel Bank Common Lisp (SBCL) uses a generational mostly-copying collector named *gencgc*. The collector also must conservatively scan the registers and stacks of the mutator when using the x86 and x86-64 instruction sets (similar to Barlett's mostly-copying collector [4]). The heap is first split into the *static*, *dynamic* and *immobile* spaces (the last only when using the x86-64 instruction set), with only the dynamic and immobile spaces ever garbage collected. Almost all objects are allocated into the dynamic space. The dynamic space is split into 32 kibibyte pages; a page may either be used for storing many small objects, or for storing part of a large object. Small objects are stored contiguously in pages. gencgc reclaims memory by copying small objects into empty pages, which causes the live objects to occupy fewer pages, and by marking large objects and pages encountered as live without copying.

There are two main inefficiencies with this approach: copying objects may require more time than marking, and the collector only uses a single core. The latter can lead to Common Lisp programs exhibiting poor scalability, to no fault of the programmer. For example, one parallel fuzz tester is *embarrassingly parallel* as tasks do not share any resources; in a tight heap, the fuzz tester runs 55 seconds of processor time over 12 cores in 5.0 seconds of real time, but the collector runs for 10.5 seconds of real time, causing the program to run for 15.5 seconds of real time, with only a 4.5× speedup.

While it is possible to improve the throughput of tracing garbage collection by increasing the size of the heap [1], it can be undesirable to use a heap much larger than the live objects, and it is not sustainable to need to use more memory in order to maintain scalability. In order to hold the impact of a serial garbage collector constant, the heap size may need to grow quadratically with regards to the number of cores used. Suppose a program processes n tasks in parallel, each task allocating r words per second and keeping m words live at any time. The system performs a garbage collection after allocating t words. A full tracing garbage collection[1] thus requires tracing nm words, and a garbage collection occurs nrt^{-1} times per second. The overall cost of tracing (which often dominates) is thus proportional to n^2mrt^{-1}. Maintaining a constant cost of collection while increasing n requires increasing t by a factor of n^2, i.e. allowing the collector to use space proportional to the square of the number of tasks to run. Assuming perfect scalability of the collector, a parallel collector can instead use n cores for tracing and a heap only n times larger, to achieve the same effect.

Increasing the heap or nursery size also decreases locality of reference, which can decrease the performance of functional programs overall [12]. However, Common Lisp programmers vary in their use of functional or in-place algorithms, and thus the significance of locality of reference may not be as large as with more functional languages.

We can also use parallelism to improve the latency of garbage collection. While our collector still stops the world in order to perform a collection, parallel garbage collectors take less real time to collect than non-parallel collectors, thus decreasing pause times.

A reader unfamiliar with garbage collection may want to consult the Memory Management Reference[2] for definitions of unfamiliar terms in this paper.

2 PRIOR WORK

Luís Oliveira parallelised gencgc [18], using an approach like that used by Marlow et al for the Glasgow Haskell Compiler [16]. In both collectors, worker threads claim pages to scan for references to objects which need to be copied. Oliveira did not achieve a large speedup by parallelising garbage collection, but he was only able to test on a dual-core machine. We suspected greater speedups could be achieved with more cores, as processors with more cores are much more accessible than at the time of development of either collector; the Steam hardware and software survey results[3] indicate that more than half of participants now have 6 or 8 cores.

[1] We believe the same relation would hold for a generational collector, but the cost model would be more complicated.
[2] https://www.memorymanagement.org/
[3] The survey results are accessible at https://store.steampowered.com/hwsurvey; the participants are users of the Steam game distribution service, so it is possible that the results are biased towards more powerful computers.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'23, Apr 24–25 2023, Amsterdam, Netherlands
© 2023 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.7816398

We attempted to replicate Oliveira's collector in 2022, but the collector did not work reliably, nor did it achieve any substantial speedup when it did work. Collector threads contended heavily on a lock while acquiring new pages to copy objects into; Marlow et al had experienced similar behaviour, and reduced the frequency of locking by having collector threads acquire more pages at once. We instead opted to implement a non-moving collector, eliminating altogether the need to support fast allocation by many threads.

3 MARK-REGION COLLECTION

3.1 Heap structure

The heap structure is based on the two-layer structure of Immix [8], consisting of 32 kibibyte *pages* consisting of 128 byte (or 16 word) *lines*. The page size may be changed, but the scavenging algorithm constrains the line size, as described in Section 4. As with gencgc, pages may either store small objects or part of a large object, but objects larger than three quarters of a page may reside on a single "large" page, to prevent the allocator from trying to search for large holes in small pages.

The collector reclaims memory at the granularity of lines; a line is considered either entirely live and not reusable, or entirely dead and reusable. The collector also marks all lines that an object occupies when tracing the object, ensuring live lines are not reclaimed later. As objects allocated together tend to die together [24], the garbage collector is still effective at reclaiming memory despite this inaccuracy, and the heap produces little internal fragmentation. The mutator allocates objects contiguously into unused lines, providing for locality of reference between objects allocated contemporaneously.

The collector relies on four additional tables stored outside of the heap: object map and mark bitmaps, and line metadata and card table bytemaps, each $\frac{1}{128}$ of the heap size on 64-bit platforms, leading to approximately 3.1% space overhead. As objects are aligned to two words (or 16 bytes), it is only necessary to store a bit for locations spaced 16 bytes apart. A byte consists of eight bits, so the locations for 128 bytes of heap fit in one byte of bitmap. The bytemaps are sized to have the same scale (of metadata bytes to heap bytes) as the bitmap, to simplify traversals of the heap, as described in Section 4.

3.2 Tracing

The collector has four stages. The collector first marks the *roots*, such as local and global variables. Then the collector *scavenges* objects in older generations to find references to new objects; such references require the collector to retain those new objects. The collector then *traces* the heap, marking every object which is transitively reachable from a marked object. The collector finally *sweeps* the heap, resetting the internal state of the collector and allowing the memory used by dead objects to be reused. The scavenging and sweeping passes can be parallelised by giving each collector thread its own section of the heap to process, but it is not as trivial to parallelise the tracing pass.

Parallel tracing is performed using the design by Ossia et al [19]. Grey objects (which have already been marked, but need to be traced by the collector) are stored in a set of *grey packets*, with each packet storing a sequence of references to grey objects. Each worker thread has an *input packet* of objects that the thread is going to trace, and an *output packet* of objects that were discovered by the thread during tracing. As collector threads read and write packets entirely sequentially, threads may use software prefetching to avoid waiting for objects to be loaded from main memory. We store packets in a stack, using a lock to protect the stack; Ossia et al used a lock-free list, but we did not observe a significant decrease in performance by using a lock. Locking has been also used in other well-used collectors; the Garbage-First collector for Java[4] used a lock, suggesting that locking the stack does not impact performance in Java.

The collector allocates packets outside the heap, directly acquiring memory using the `mmap` system call.[5] In order to avoid serialisation induced by the kernel updating the memory map, we implemented an arena allocator, which allocates chunks of packets sized to increasing powers in two. At the end of the collection, the entire contents of the arena are considered unused. In order to avoid allocations in subsequent collections, we retain chunks which have been used recently, but we `munmap` chunks which have not been used recently, so that the collector does not needlessly retain chunks which are seldom used. To avoid having to protect a global free list from concurrent access, packets are reused in thread-local lists. We confirmed Ossia et al's observation that packets use little memory, at most using 3.7 MB while running the boehm-gc benchmark in a 4 GB heap (as described in Section 6).

4 NON-MOVING GENERATIONAL COLLECTION

Many approaches to *generational* garbage collection rely on representing generations using ranges of addresses in the heap. For example, gencgc associates a generation with each page in the heap. We avoid moving objects where possible, so attempting to partition the heap using addresses prevents the collector from reclaiming much memory; if a page were promoted to an older generation, any free space on the page could not be used for allocating new objects (which necessarily are in the youngest generation), and could not be reused until the page is entirely unused.

Demers et al suggested that it is not necessary to represent generations this way, however, and that associating a generation with each object suffices [9]. We are left with the problem of where to store the generations associated with each object. All types of objects that are not cons cells have a *header* word in SBCL, which can store a generation number[6], but there is no free space in a cons cell to store a generation number. We may instead store generation numbers in a table external to the heap, as we do with the mark bitmap. Using a table separate from the heap also reduces the amount of memory which must be scanned, as generation numbers are stored compactly, instead of being mixed with other data in the heap. Scanning the table also eliminates the need to walk most objects in the heap.

[4]See https://github.com/openjdk/jdk/blob/jdk-21+8/src/hotspot/share/gc/g1/g1ConcurrentMark.cpp#L175-L211
[5]As the collector runs in a signal handler, it is necessary to avoid using libc functions, and SBCL generates a "raw" system call when possible.
[6]Indeed this is how the *immobile space* in SBCL works, as it does not need to store cons cells.

Making such a table space-efficient would be difficult, however, as a generation number would be required for every location that an object could reside at. SBCL uses eight generations by default, requiring three bits of storage per location. If a full byte were reserved for each generation number, the generation table would require a $\frac{1}{16}$ space overhead. Immix peculiarly used a table of bytes (which we will call a *bytemap*) for storing marks for lines, rather than a table of bits (a bitmap), in order to avoid using atomic operations to mark lines as live. We may instead use the bytes in the bytemap to store generations and line marks; as we only collect whole lines, parts of a line cannot be reused, so all objects on a line must share the same generation. To implement conservative root finding, the line bytemap is also used to record which lines have been freshly allocated.

Demers et al also raised the issue of *card pollution*. Generational collectors often use a card map [23] in order to find old objects which have been updated, and may now contain references to new objects. Cards are segments of the heap, similar to pages, although cards are usually smaller than pages. The compiler inserts *write barriers* into mutator code, which cause the mutator to mark a card as *dirty* when the mutator stores a reference in that card. If cards are larger than lines, then newer and older objects may exist in the same card. It is not possible for the collector to discern writes to newer and older objects on the same card, so writes to newer objects in a card may cause the collector to needlessly scavenge older objects in the same card. We avoid this imprecision by making cards the same size as lines, so that cards can only store objects in the same generation.

Another benefit to making cards the same size as lines is that various operations on all the bitmaps can be performed more efficiently using *single instruction-multiple data* instructions, which many instruction sets have. It is convenient to use SIMD comparison instructions on the bytemaps, as *true* is represented as all bits set in a byte, and *false* as all bits cleared[7]. Such results can be treated as sets of object locations where tests on the line and card bytemaps succeed, and bitwise-and may be used to perform logical conjunction of tests. For example, code computing $map \land (generation > g) \land (card = dirty)$ for every byte of the object map, line bytemap and card bytemap will correctly compute a bitmap with bits set where objects which are dirty and are older than generation g reside.

5 CONSERVATIVE ROOT FINDING

The SBCL compiler does not record which registers and stack locations are used for tagged and untagged values, when targeting the x86 and x86-64 instruction sets, so a collector must be *conservative* when scanning the stack and registers to find root references into the heap; the collector must be able to identify which values identify objects in the heap, and which do not. While SBCL does not use *interior pointers*, which keep an object live without pointing to the start of the object, we implemented interior pointers to see how complicated implementing interior pointers would be.

[7]For example, the SSE2 instruction `pcmpeqb` effectively computes `(map 'vector (lambda (a b) (if (= a b) #xFF #x00)) A B)` for two vectors A and B. In practise we rely on the auto-vectorisation of C compilers for portability; GCC and Clang successfully generate vectorised code for x86-64 (with SSE2 and AVX2), ARM (with SVE) and RISC-V (with the Vector extension).

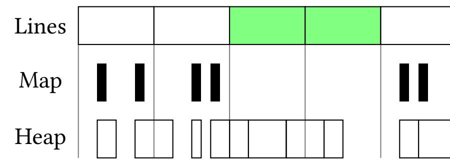

Figure 1: Most objects are not contiguous in memory, but objects allocated after the last collection are contiguous in *fresh* lines (light green).

If objects are stored sequentially in pages, the collector can scan a page of memory sequentially to find the object which a pointer references [5]; SBCL currently uses this approach. We cannot use this approach, however, because objects are not stored sequentially in pages, and so attempting to scan sequentially will likely read garbage data.

Another approach is to record positions of objects into an *object map* bitmap when allocating, as suggested by Shahriyar et al [21]. Without interior pointers, it suffices to check the bit in the object map corresponding to a pointer, to determine if the pointer points to an object in the heap. With interior pointers, the bitmap must be scanned backwards to find a set bit, and then the size of the corresponding object found may additionally be checked, in order to confirm that the pointer does indeed point inside the object. The scan is rather fast when employing *bit parallelism* and checking entire words of bits at a time. However, setting bits in the bitmap slows down the mutator, and it is also difficult to update the bitmap without dedicated instructions; Shahriyar et al use the x86 `bts` instruction to set a bit in a bitmap, but many other instruction sets such as ARM and RISC-V do not have a similar instruction.

We instead use a hybrid of the two approaches, where an object map is used, but it is not updated by the mutator directly. While Immix does not store objects contiguously on pages, objects are allocated contiguously in smaller runs. An example of this behaviour is depicted in Figure 1. Our allocator marks lines it uses for allocation as *fresh*, and when the collector encounters a conservative reference into a fresh line, the collector finds the start and end of the enclosing run, and computes the object map for that run of objects. As most objects die young [22] and few objects are referenced from the stack, the collector does not have to compute much of the object map. It is also thus not necessary to produce a fast instruction sequence to update the object map, as the object map is seldom written into, and the object map is not written into by the mutator. We have observed that, at most, the parallel fuzz tester requires the collector to compute the object map for about 600 kilobytes of heap on average.

This approach might increase pause times, as object map computation is done all at once, rather than spread out across the execution of the application. We have not observed any effect on pause time in practise, but a concurrent collector may opt to process conservative roots concurrently with the mutator.

6 PERFORMANCE

We improve performance by using multiple cores to perform garbage collection work, but it is also important that the performance with a single core is not made much worse by the use of a more scalable

parallel algorithm. Using many more cores for a small performance gain would lead to very poor efficiency, which is usually undesirable. Focusing on only using more cores to achieve performance is likely futile in any case, as the scalability of tracing is often limited by the shape of the heap being collected [3]. Thus we report results for a variety of thread counts: we test the baseline performance of the collector with only one thread, realistic configurations with 2 and 4 threads, and the limit with 12 collector threads. We test with a Ryzen 1950X processor, with 12 cores provided to the virtual machine used. Each configuration of each benchmark was run 30 times, and we report the average of all the runs.

We test with commit `4e71abc577180c0276cb14b31a69dc0d2eb84694` of our fork of SBCL accessible at https://github.com/no-defun-allowed/swcl, with the immobile space disabled for both collectors, as we currently cannot use it with the parallel collector. Enabling the immobile space makes the mutator much faster running Regrind; we expect a similar speedup would be achieved when the immobile space works with the parallel collector.

We use our own benchmark suite as we could not find any other suite which was appropriate. In particular, the *cl-bench* suite is popular, but does not generate much of a load for the garbage collector. We could decrease the size of the heap in order to cause more frequent garbage collections, but more of the heap would still fit in the large caches of modern processors. There are also no latency-sensitive benchmarks in cl-bench, nor any benchmarks which use multiple threads. The benchmark suite itself is accessible at https://github.com/no-defun-allowed/gc-benchmarks/tree/v1.

6.1 Throughput benchmarks

Figure 2 contains the results of the throughput benchmarks. The benchmarks are:

- **boehm-gc**: A benchmark from *cl-bench*[8] which allocates many binary trees of various sizes, with the k parameter (which affects the size of trees allocated) increased to 24 from the original default of 18. Binary trees of each stage of the benchmark die simultaneously, and fragmentation is negligible. Serial mark-region collection lags copying collection somewhat, and any parallel collection out-performs gencgc. The benchmark also runs in a smaller heap when using the mark-region collector.
  Despite the formerly discussed efforts to reduce mutator overhead, some overhead still exists when using the mark-region collector. We suspect that the smaller cards cause more cache misses, slowing down the mutator.
- **regrind-interpret**: The parallel fuzz tester for the *one-more-re-nightmare* regular expression compiler[9], running using 12 worker threads. In order to make the benchmark more reproducible, it was modified to generate the same sequence of regular expressions to test, and to use static load balancing. It allocates lots of very short-lived objects. gencgc is comparable in performance to parallel mark-region collection with two threads, owing to the lower mutator time. The mark-region does not perform well with a heap smaller than 5GB, and all but the 12-thread configuration are outperformed by gencgc with a heap larger than 5GB. Very few objects survive a nursery collection, causing the scavenging and sweeping passes of the mark-region collector to take most of collection time.
  This benchmark exhibits a more asymptotic mutator performance than *boehm-gc*, with the mutator performance varying with the heap size when using mark-region collection. The performance appears to vary due to the mutator needing to acquire more pages in tight heaps (as in Figure 3). A partly used page fits fewer new objects than an entirely free page, so more partly filled pages must be acquired to allocate; boehm-gc does not produce any partly used pages. While the collector can reuse partly used pages without moving, it incurs some time overhead in doing so. (We thus have lost some scalability to the allocator, but it is not as bad as if we had used a parallel copying collector.)
- **regrind-compile**: The same fuzz tester, with the same modifications made as with regrind-interpret, but using the compiler of SBCL rather than the interpreter. (Using the compiler is much slower than the interpreter, so the fuzz tester is configured to perform much less work.) It allocates longer-lived objects, requiring much more time to trace. The variation of mutator time is greater than in regrind-interpret.

6.2 Latency benchmarks

One benchmark we test, named **ring-buffer** is pathological for copying collectors [13], demonstrating a substantial difference in work performed by non-moving and copying collector algorithms. The benchmark involves a ring buffer of small unboxed arrays, each array containing a one kilobyte "message", with each new message being a new allocation from the heap. Each live message must be copied by a copying collector, although no pointers to trace are discovered in doing so.

As depicted in Figure 4, the mark-region collector performs much better by not having to copy messages, and also allows running in smaller heaps. When running in a 2GB heap and with gencgc, the program spends 50% of processor cycles in the C `memcpy` function. While the benchmark is derived from a real program which had this pathological behaviour, we do not think it is a good model of most programs. Almost all objects allocated in the benchmark have no pointers, are somewhat large, and survive several nursery collections.

Kandria, a commercial game written in Common Lisp [14], is used in a more complex benchmark. The game uses a mixture of object-oriented code and numerical code with unboxed arrays. The game is configured identically to the retail version, using a 4GB heap. We played the same part of the game for a few minutes with each collector configuration[10], and record the distribution of how long it takes to produce each frame (*frame times*), and the distribution of pause times in the **kandria** benchmark. As the game

[8] https://gitlab.common-lisp.net/ansi-test/cl-bench/-/blob/master/files/boehm-gc.lisp
[9] https://github.com/telekons/one-more-re-nightmare/blob/master/Tests/regrind.lisp

[10] We would prefer to be able to replay a *capture* of inputs to the game, in order to have the game run more deterministically, but we were not able to get the capture to be replayed reliably.

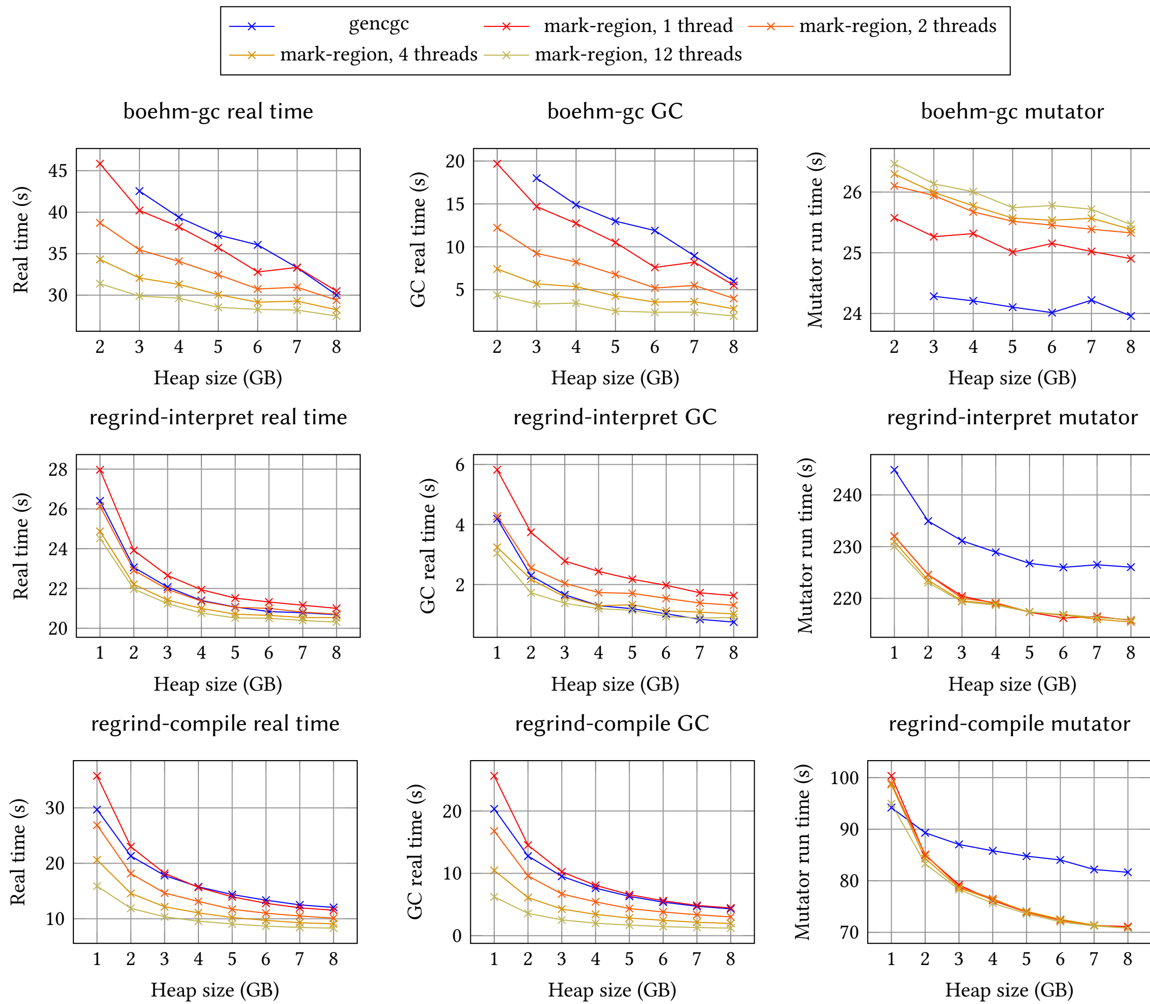

Figure 2: Results of the throughput-oriented benchmarks.

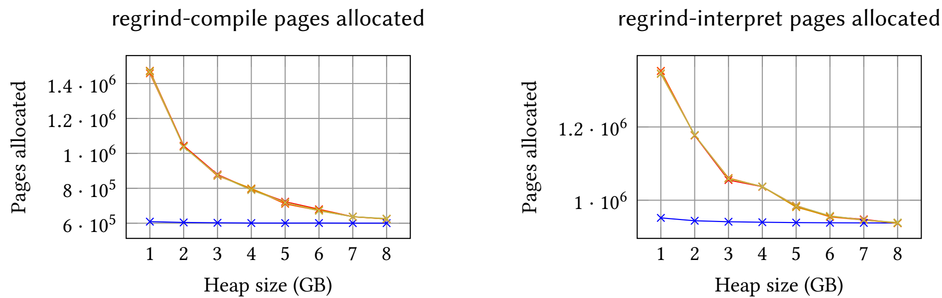

Figure 3: The number of pages acquired in order to allocate small objects.

requires low-latency graphical input and output, Kandria was run on the author's desktop computer, with a Ryzen 5900X processor and RX 580 graphics card.

As depicted in Figure 5, parallel garbage collection slightly reduces the size of the tail of pause times. However, the parallel collector does not improve pause times substantially. We also compared frame times to some time limits (in Table 1): the 16ms time limit tests if the game can run smoothly at 60 frames per second, and shorter time limits approximate the same target while using a

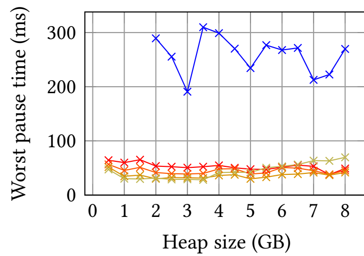

Figure 4: Worst pause times running ring-buffer.

Table 1: The percentages of frame times which exceed various time limits, for varying collectors and thread counts. (Mark-region is abbreviated to MR.)

Limit	gencgc 1 thread	MR 1 thread	MR 2 threads	MR 4 threads
16ms	0.22%	0.28%	0.27%	0.26%
12ms	0.28%	0.38%	0.35%	0.34%
8ms	0.43%	0.52%	0.50%	0.51%
6ms	2.15%	2.30%	2.21%	2.23%

slower computer. The mark-region collector violates the time limits more often than gencgc, regardless of the number of threads used.

Very few objects survive garbage collection in Kandria, with less than a megabyte surviving out of a 200 megabyte nursery. The scavenging and sweeping passes dominate collection time, due to doing work proportional to the number of used pages. We observed at one point that there were older objects occupying 160MB of memory spread across lines occupying 310MB of memory, in turn spread across pages occupying 660MB of memory. This fragmentation causes the collector to scan much more metadata than is strictly necessary. As a result, scavenging took 2.9 milliseconds on average, tracing took 1.9 milliseconds, and sweeping took 1.6 milliseconds. Kandria thus appears to represent a pathological case for non-copying collectors.

7 CONCLUSIONS AND FUTURE WORK

When using a single core, our mark-region garbage collector is only somewhat slower than the copying collector of SBCL, despite our collector never moving any objects. Using additional cores to collect in parallel allows our collector to significantly outperform the copying collector, and non-moving collection appears to be simpler to correctly parallelise than copying collection.

The collector is not ready to be used in production yet, lacking support for the immobile space of SBCL, and lacking any kind of compaction. We are also considering extending the collector to operate *concurrently* with the mutator, which is simpler without needing to copy objects.

7.1 Immobile space

SBCL has an additional *immobile space* which resides in the lowest 2^{32} bytes of the address space, and does not move. The immobile space stores layouts of "instance" objects to reduce the size of the headers of instance objects, and stores symbols to reduce the size of code referencing symbols. The immobile space is managed by a different marking algorithm, and by the TLSF allocator [17] rather than a bump allocator. We haven't succeeded in getting the immobile space collector to work with the mark-region collector yet, but it should be used in a garbage collector used in production.

As the parallel collector used for most of the heap (in *dynamic space*) does not move, it is tempting to simplify the heap and use the same collector for immobile space. But allocations into immobile space are infrequent, and objects in immobile space can never be compacted (except when saving a core file), so it is worthwhile to proactively avoid fragmentation by using the more complex TLSF allocator.

7.2 Compaction

Heap fragmentation, due to our collector not moving objects, leads to more pages being used than necessary. While the SBCL process can effectively reuse holes in pages, the space is unusable by other processes on the same computer. The lack of compaction also affects the size of *core files*; for example, the `sbcl.core` core file using the copying collector is 36 megabytes large, but the file is 248 megabytes large using the mark-region collector, as the mark-region collector never compacted the heap during bootstrapping. Compaction can also coalesce free space into full pages, reducing the number of pages that the mutator must acquire. Compaction may help to regain some locality of reference, if objects are compacted into fewer pages. Compaction can also reduce the amount of metadata that scavenging and sweeping need to scan, which may be particularly useful for Kandria.

We have begun to implement an algorithm for *incremental* compaction [6] which only moves part of the heap at the time. We select pages with few lines used starting from the end of the heap, and copy their contents into holes in the start of the heap. Collector threads record references to the selected pages while tracing, so that those references can be fixed up after copying has been performed. Unlike the mixture of marking and copying that Immix performs in one pass, having separate passes allows marking to be performed concurrently, while compacting is done in a (hopefully shorter) stop-the-world pause.

The algorithm may not perform well with many generations, however, as we cannot identify all references to older objects when performing a collection of a younger generation, and so we cannot move older objects correctly. It may be helpful to analyse if the many generations used by SBCL are beneficial; many generational garbage collectors with good performance only use a young generation and an old generation, and all objects could be moved when collecting the old generation with just two generations.

7.3 Concurrent tracing

The collector could be made concurrent by following the Ossia et al design. The Ossia et al collector did not support generations, but used the card map to detect modifications of any object while the collector is tracing. In order to support generations, another card table storing only the locations of old-to-new references would need to be maintained by the collector. Both card tables would need

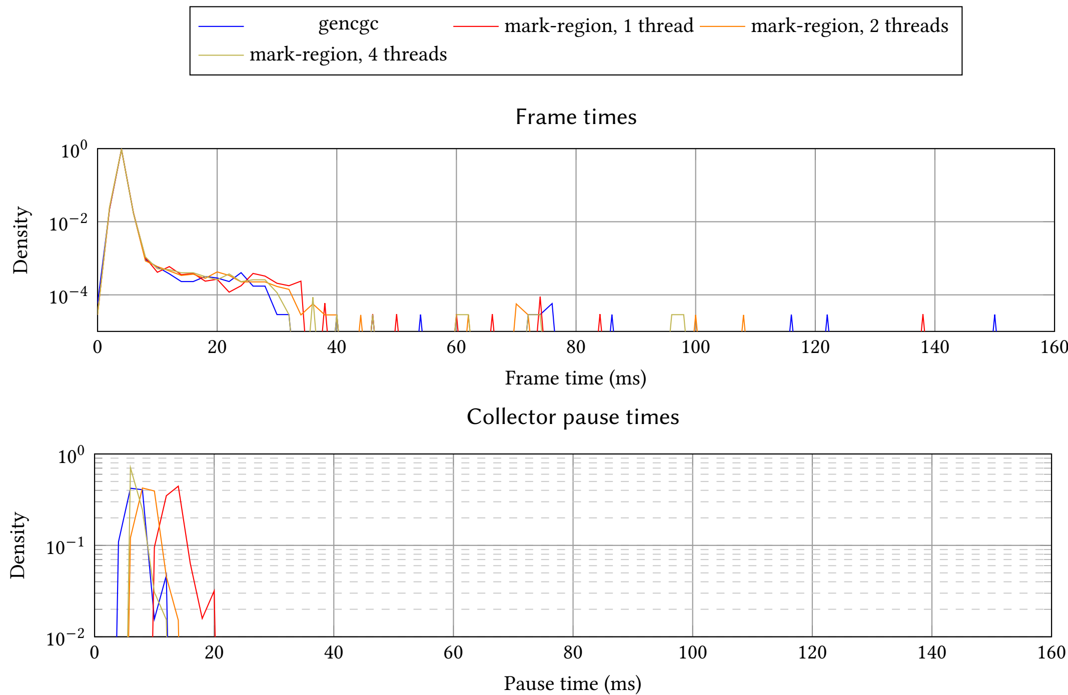

Figure 5: Frame times and collector pause times running Kandria.

to be consulted to find all old-to-new references for collection of younger generations.

It is possible to make compaction concurrent, typically by using a *read barrier* [2], which allows for ensuring the mutator only accesses pointers to copied objects, but using a read barrier induces more time overhead on the mutator. Recently Zhao et al have suggested that the latency which a user of the application experiences is better served by improving throughput, rather than focusing on further minimising pause times [25]. As we intend to compact infrequently, application latency may not be greatly affected by compaction pauses.

7.4 Other ways to collect

There are other approaches to make garbage collection more performant on multi-core computers, which we believe should be reconsidered. For brevity we will only discuss *reference counting* and *thread-local garbage collection* in some depth.

While reference counting has been seen as inferior in performance to tracing, it has been optimised with *coalescing* [15] to greatly reduce the number of updates to reference counts, combined with the Immix heap layout to provide better locality of reference [20], and recently the LXR collector [25] has been shown to outperform other garbage collectors for Java in both throughput and latency. Updating reference counts in a coalescing reference counting collector can be embarrassingly parallel, unlike tracing. Reference counting cannot collect cycles however, so infrequent tracing to collect cycles is necessary; but if tracing is infrequent, it will not harm scalability too much. It has also been observed that old objects are less often modified than young objects in Java [7]; if a similar observation holds for Common Lisp programs, using coalesced reference counting to reclaim old objects may work well, with fewer updates to reference counts contributing to a stop-the-world pause.

Another approach to improving the scalability of tracing is to use *thread-local* garbage collection, for which designs for immutable objects in ML [10] and mutable objects in Java [11] exist. Each thread has its own private nursery, which may be collected independently and without synchronisation, allowing for high scalability. Thread-local collections also may improve latency, as *global* collections which require all threads to be stopped are less frequent. Locality of reference may also be improved, as thread-local nurseries can be small; and cache ping-pong effects are minimised, as threads never need to access cache lines for the nurseries of other threads.

The latter design, which does not require objects to be moved, could benefit from a mark-region heap as described in this paper; in particular, the mutator can still utilise bump allocation, even though global objects cannot be moved out of partly used pages without performing a global collection. A similar kind of sweeping can be used for sweeping during a local collection, by copying the mark bitmap to the object bitmap only for local objects, thus preserving global objects which are not collected.

ACKNOWLEDGMENTS

We would like to thank Stas Boukarev and Douglas Katzman for helping us get up to speed with the interactions between the garbage collector and the rest of SBCL. Steve Blackburn and Kunal Sareen had discussed Immix with us, leading to the approaches to implementing generations and conservative root finding described in

this paper. Nicolas Hafner helped us get auto-vectorisation to work, and helped us navigate the Kandria source code. Paul Khuong and Larry Masinter helped us design the internal memory manager used by the garbage collector. Cliff Click helped us make sense of our benchmark results with parallel programs. Grindwork Corporation provided us with access to a virtual machine for benchmarking. Jan Moringen, Kunal Sareen, Elijah Stone, and Robert Strandh provided comments on early versions of this paper.

REFERENCES

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25(4):275–279, June 1987. ISSN 0020-0190. doi: 10.1016/0020-0190(87)90175-X.

[2] Henry G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, April 1978. ISSN 0001-0782. doi: 10.1145/359460.359470.

[3] Katherine Barabash and Erez Petrank. Tracing garbage collection on highly parallel platforms. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, page 1–10, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300544. doi: 10.1145/1806651.1806653.

[4] Joel F. Barlett. Mostly-copying garbage collection picks up generations and C++. Technical report, Digital Western Research Laboratory, 1989.

[5] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. *SIGPLAN Lisp Pointers*, 1(6):3–12, April 1988. ISSN 1045-3563. doi: 10.1145/1317224.1317225.

[6] Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, page 100–105, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135394. doi: 10.1145/512429.512442.

[7] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, page 344–358, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137125. doi: 10.1145/949343.949336.

[8] Stephen M. Blackburn and Kathryn S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 22–32, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375586.

[9] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 261–269, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913434. doi: 10.1145/96709.96735.

[10] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL 1993: 20th symposium Principles of Programming Languages*, pages 113–123. ACM, 1993. doi: 10.1145/158511.158611.

[11] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. *SIGPLAN Not.*, 38(2 supplement): 76–87, Jun 2002. ISSN 0362-1340. doi: 10.1145/773039.512439.

[12] Henrique Ferreiro, Laura Castro, Vladimir Janjic, and Kevin Hammond. Kindergarten cop: Dynamic nursery resizing for GHC. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 56–66, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342414. doi: 10.1145/2892208.2892223.

[13] Jim Fisher. Low latency, large working set, and GHC's garbage collector: pick two of three, 2016.

[14] Nicolas "Shinmera" Hafner. Experience report: Kandria - a game in Common Lisp. *The 16th European Lisp Symposium (ELS'23)*, 2023.

[15] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, jan 2006. ISSN 0164-0925. doi: 10.1145/1111596.1111597.

[16] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, page 11–20, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581347. doi: 10.1145/1375634.1375637.

[17] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, ECRTS '04, page 79–86, USA, 2004. IEEE Computer Society. ISBN 0769521762.

[18] Luís Oliveira. SBCL garbage collection. Master's thesis, Universidade de Coimbra, 2009.

[19] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. *SIGPLAN Not.*, 37(5):129–140, May 2002. ISSN 0362-1340. doi: 10.1145/543552.512546.

[20] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S. McKinley. Taking off the gloves with reference counting Immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 93–110, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323741. doi: 10.1145/2509136.2509527.

[21] Rifat Shahriyar, Stephen M. Blackburn, and Kathryn S. McKinley. Fast conservative garbage collection. *SIGPLAN Not.*, 49(10):121–139, October 2014. ISSN 0362-1340. doi: 10.1145/2714064.2660198.

[22] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Not.*, 19(5):157–167, April 1984. ISSN 0362-1340. doi: 10.1145/390011.808261.

[23] P. R. Wilson and T. G. Moher. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *SIGPLAN Not.*, 24(5):87–92, May 1989. ISSN 0362-1340. doi: 10.1145/66068.66077.

[24] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, page 1–116, Berlin, Heidelberg, 1995. Springer-Verlag. ISBN 3540603689.

[25] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. Low-latency, high-throughput garbage collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 76–91, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523440.