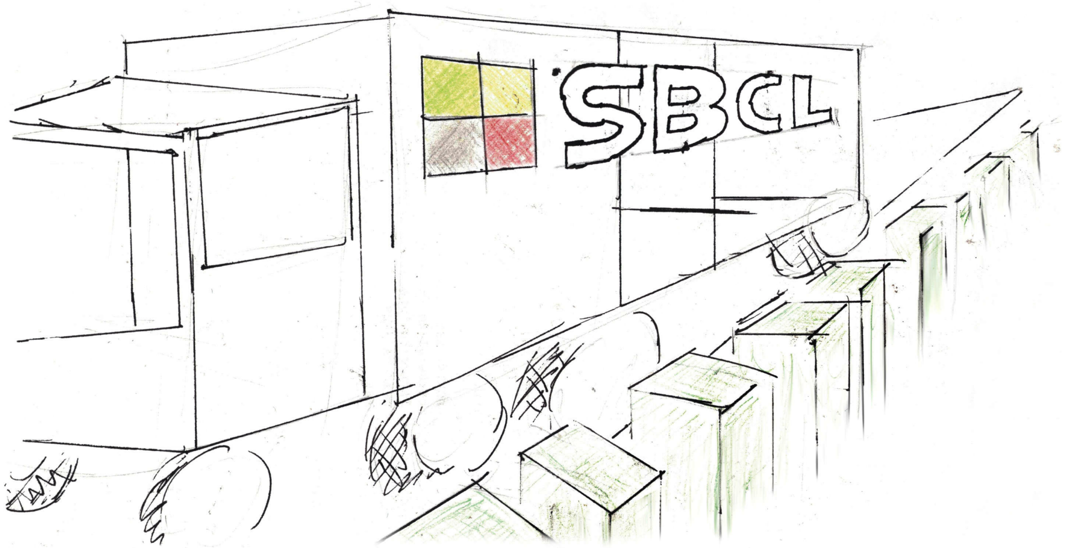


Parallel garbage collection for SBCL

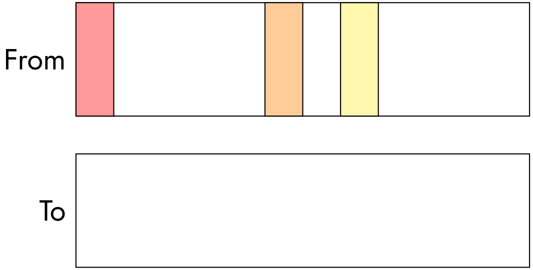
Hayley Patton, 25 April, 2023

Outline

- I've implemented a new garbage collector for SBCL.
- The collector can use multiple cores, and it can sometimes run faster by not copying objects.
- It uses new approaches to conservative root finding and non-moving generational garbage collection.
- It should still move objects sometimes, which is part of why it's not in upstream yet.
- But it also should be easier to make (mostly) concurrent, as it doesn't need to move objects too frequently.

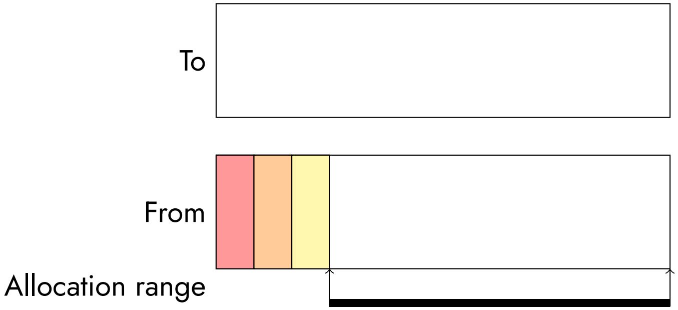
The current SBCL collector

SBCL uses a **gen**erational **c**onservative **g**arbage **c**ollector (gencgc). It reclaims most memory by copying the live objects, leaving large spans of free space.
A simple copying collector copies objects from one space to another, with each *semi-space* being half the size of the heap. The application can allocate objects by incrementing a pointer (*bump allocation*), which starts at the end of the last object.



The current SBCL collector

SBCL uses a **gen**erational **c**onservative **g**arbage **c**ollector (gencgc). It reclaims most memory by copying the live objects, leaving large spans of free space.
A simple copying collector copies objects from one space to another, with each *semi-space* being half the size of the heap. The application can allocate objects by incrementing a pointer (*bump allocation*), which starts at the end of the last object.

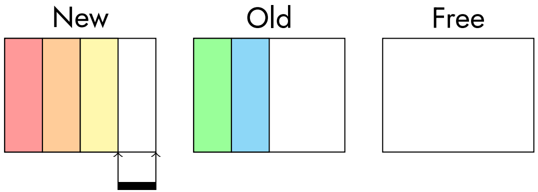


The current SBCL collector

SBCL doesn’t use two contiguous spaces however; instead it partitions memory into *pages*, which are assigned to spaces using a table. Pages are also assigned to different generations and types.

The application bump allocates until the allocation pointer reaches the end of a page.

The application finds a suffix of another page to allocate into, after using all the space in a page.



Strengths and weaknesses of gencgc

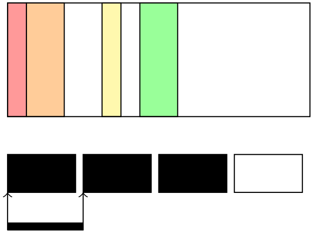
- **Strength**: Copying avoids *fragmentation*.
  But real applications don't fragment that much, so it's mostly wasted effort.
  Copying also requires "wasting" space to copy into.
- **Strength**: Bump allocation is simple and fast.
  But we can bump allocate with other algorithms.
- **Weakness**: Copying requires more memory accesses than other algorithms which don't move objects.
- **Weakness**: Parallel copying requires careful synchronisation, and copying might hit memory bandwidth limits with fewer threads.
  But if there are few live objects (common in generational GC), copying does less work than algorithms which scan the whole heap somehow.

Workarounds

- We could increase the size of the heap, to make the garbage collector run less frequently, to improve throughput. But this doesn't scale when parallelising a program. Also locality of reference concerns.
- Increasing the size of the heap also makes the time between pauses longer, but doesn't shorten pauses, which doesn't help latency much.
- Avoiding generating garbage tends to hinder modularity and simplicity; "liveness is a global property."

Mark-region garbage collection

- We instead implement a *mark-region* collector, similar to *Immix*. The collector can reclaim memory without moving objects, and the application can still bump-allocate.
- Pages are partitioned again into smaller *lines*, each 128 bytes, which the collector can individually reclaim. The application bump-allocates into runs of free lines.
- The collector traces through every live object in-place, and then reclaims lines unused by any live objects.



Tracing

- Tracing recursively visits all objects which the application could possibly access. We speak as if objects have colours: objects not visited are *white*, objects visited but not recursively traced yet are *grey*, objects which were recursively traced are *black*.
- A semi-space collector represents the grey and black sets using contiguous ranges of to-space. (*Cheney's algorithm*; it's somewhat trickier when using pages, but the basic idea remains.)
- Other tracing collectors require the set to be represented in a different way. We instead use a list of *packets* of grey objects (following Ossia et al).
  At most this list used 3.7 MB of memory in testing.

Parallel tracing

- Multiple threads can be used to trace objects. Each thread is both a *producer* and *consumer* of grey objects.
- Each thread thus has an *input packet* of objects which the thread will trace (shading from grey to black), and an *output packet* of objects that the thread discovered during tracing (shading from white to grey). Thus threads only synchronise when getting new packets.
- Threads can also use *software prefetching* to cover up memory latency.

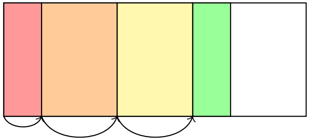
Conservative root finding

- Tracing starts with marking the objects immediately accessible by the application (the *roots*). Roots include lexical and special variables and some global data structures such as tables of packages and classes.
- SBCL can use *boxed* values, which are stored with type tags, and *unboxed* values without tags (such as 64-bit integers and `double-floats`).
- The collector must correctly handle unboxed values, despite the compiler not generating any description of which registers and stack locations have unboxed values.
- Thus the collector has to stomach unboxed integers and `double-floats` somehow.

Conservative root finding

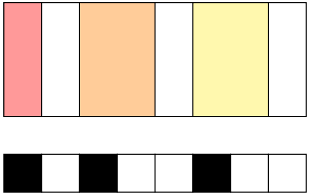
Suppose we have a “raw” value which looks like a pointer into some page.

A copying collector can step over each object until it finds the pointer, as objects are contiguous.



But our objects aren’t contiguous.

A non-copying collector can consult a bitmap which indicates where objects start.

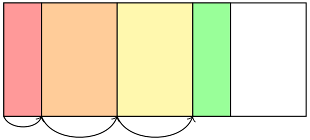


But it takes time to write into the bitmap, and requires more code outside x86.

Conservative root finding

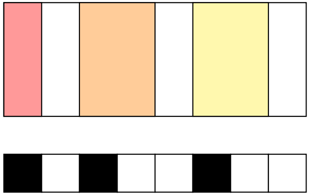
Suppose we have a “raw” value which looks like a pointer into some page.

A copying collector can step over each object until it finds the pointer, as objects are contiguous.



But our objects aren’t contiguous. Actually, objects the application just allocated *are* contiguous.

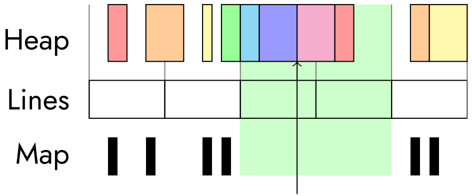
A non-copying collector can consult a bitmap which indicates where objects start.



But it takes time to write into the bitmap, and requires more code outside x86. But we don’t have to write *all* the objects.

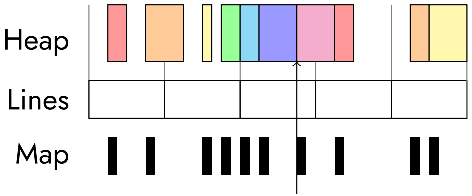
Lazy object mapping

- The allocator marks lines it just allocated into as *fresh*, and the garbage collector clears this mark during collection.
- We usually consult a bitmap to find an object, but when a raw value points into a fresh line (shaded green), we find the enclosing run of fresh lines, and compute the bitmap for that range.
- Most objects die young, and most objects aren't referenced from the registers and stack locations, so fewer bits need to be set. (At most 600 KB in testing.)



Lazy object mapping

- The allocator marks lines it just allocated into as *fresh*, and the garbage collector clears this mark during collection.
- We usually consult a bitmap to find an object, but when a raw value points into a fresh line (shaded green), we find the enclosing run of fresh lines, and compute the bitmap for that range.
- Most objects die young, and most objects aren't referenced from the registers and stack locations, so fewer bits need to be set. (At most 600 KB in testing.)

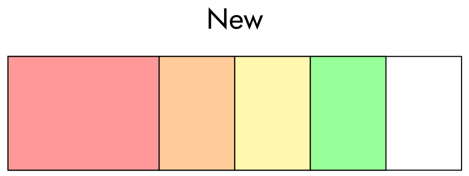


Non-moving generational collection

- Many collectors are *generational*; they separate old and new objects and collect the new objects more often.
- In practise the new objects usually die sooner, so collecting just new objects is more effective at reclaiming memory.
- But we need to partition objects by generation. The copying collector assigns each page a generation.
- We need to remember pointers from old objects to new objects, in order to collect just new objects correctly. We continue to use the *card marking* scheme, where we mark that a range of memory (a *card*) was dirtied and have GC *scavenge* old objects on dirty cards.

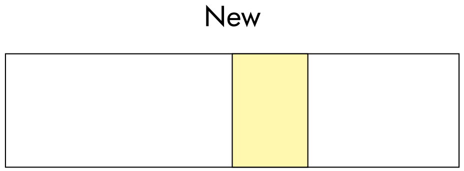
Non-moving generational collection

If we just assign each page a generation, we can't reuse any memory on an older page until *no objects* reside on it.



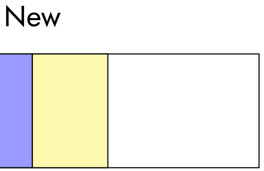
Non-moving generational collection

If we just assign each page a generation, we can't reuse any memory on an older page until *no objects* reside on it.



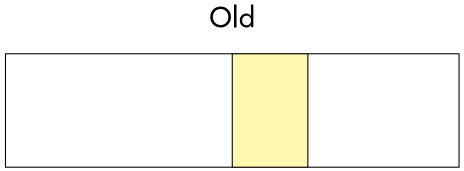
Non-moving generational collection

If we just assign each page a generation, we can’t reuse any memory on an older page until *no objects* reside on it.



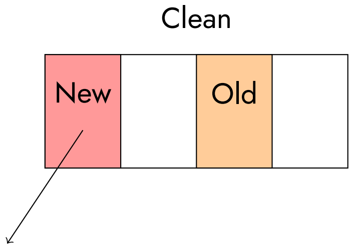
Non-moving generational collection

If we just assign each page a generation, we can't reuse any memory on an older page until *no objects* reside on it.



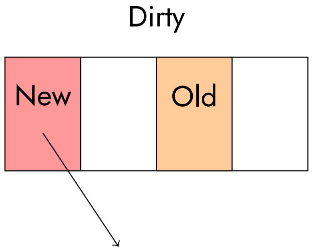
Non-moving generational collection

We could assign each object a generation, but then it becomes harder to discern writes to old and new objects on the same card. (Demers et al call it *card pollution*.) We also can't store generation IDs in objects, as there's no room in cons cells. A side table with a space for every possible object address would be quite large ($\frac{1}{16}$ of the heap on x86-64).



Non-moving generational collection

We could assign each object a generation, but then it becomes harder to discern writes to old and new objects on the same card. (Demers et al call it *card pollution*.) We also can't store generation IDs in objects, as there's no room in cons cells. A side table with a space for every possible object address would be quite large ($\frac{1}{16}$ of the heap on x86-64).



Non-moving generational collection

Solution: make lines the same size as cards.

- We don't lose any precision in reclaiming memory; we already can't reclaim parts of lines.
- We don't lose any precision in tracking updates; we can't have different generations on the same line.

Non-moving generational collection

Solution: make lines the same size as cards.

- We don't lose any precision in reclaiming memory; we already can't reclaim parts of lines.
- We don't lose any precision in tracking updates; we can't have different generations on the same line.

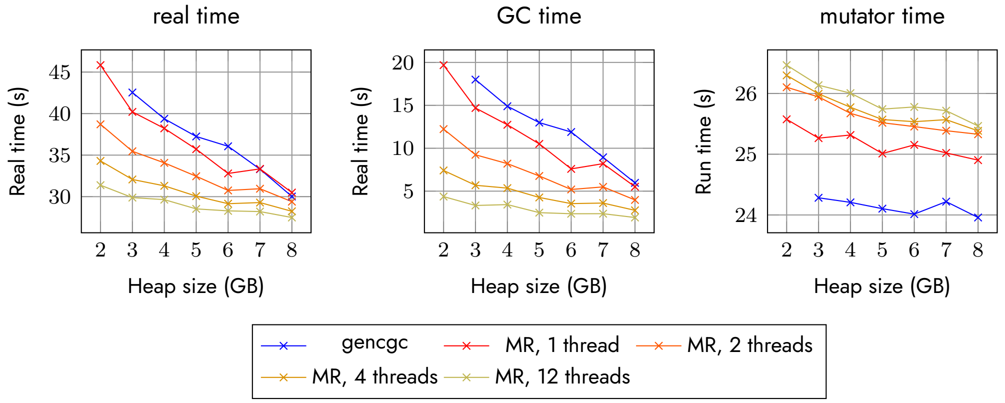
It's also very convenient to locate dirty and old cards; *single instruction-multiple data* instructions work really well here. One byte per line yields just a $\frac{1}{128}$ space overhead.

Benchmarks

- I chose perhaps rather odd benchmarks - the *cl-bench* benchmark suite doesn't generate much work for the garbage collector.
- But *boehm-gc* from the suite can be scaled easily.
- We can't really scale down the heap to introduce strain, as cache memory would become too effective to be representative of larger applications.
- So I chose two micro-benchmarks (*boehm-gc* and *ring-buffer*) and two macro-benchmarks (*Kandria* and *Regrind*).
- All benchmarks except for Kandria ran on a VM on a Threadripper 1950X with 12 cores.

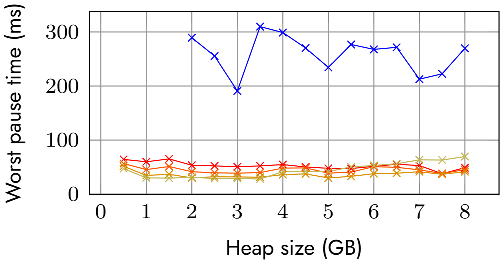
boehm-gc

boehm-gc creates many large binary trees, which the GC must trace. It produces no fragmentation, as all nodes in all binary trees die at once.



ring-buffer

ring-buffer keeps updating a ring buffer with “message” arrays, each about 1 KiB large. This is a rather unfortunate benchmark for generational copying collectors, including that of GHC[1]; as the messages live moderately long and must always be copied.



[1]https://pusher.com/blog/latency-working-set-ghc-gc-pick-two/

Kandria

Kandria is a commercial video game from Shirakumo written in Common Lisp. There were reports of stuttering on older hardware, which could be caused by GC. It generates *very* few objects which survive garbage collection, which is great for a copying collector, but not great for this mark-region collector.

(No heaps were harmed in the making of this image.)

Kandria

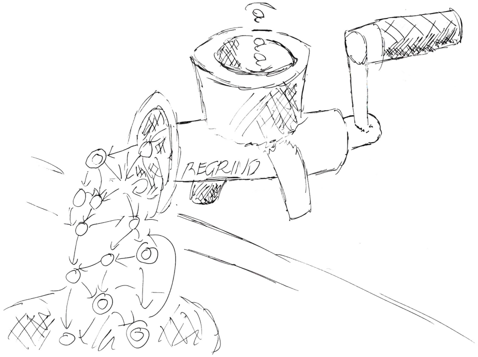
Kandria needs to be run on a computer with graphical output (in part because we couldn't get *capturing* to work, and so I had to play the game myself). I used my desktop with a Ryzen 5900X processor and RX 580 graphics card, and a 4 GB heap (as is done in the commercial distribution of the game).
Slower hardware was simulated by using tighter frame time limits.

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%

The mark-region collector suffered from fragmentation, taking 2.9 ms to scavenge, 1.9 ms to trace, and 1.9 ms to sweep on average. 160 MB of objects were spread across 660 MB of pages at one point.

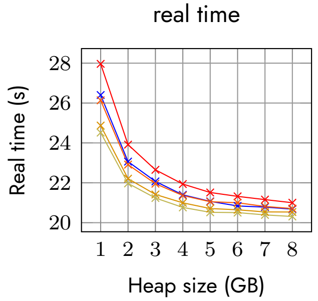
Regrind

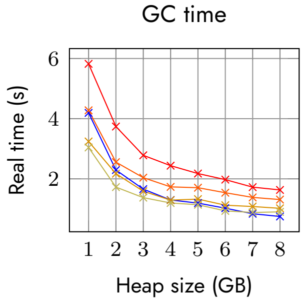
Regrind is a parallel fuzz tester for the *one-more-re-nightmare* regular expression compiler. It attempts to find cases where the compiler fails to generate an automaton, and where the compiler generates code which attempts to read out-of-bounds or returns out-of-bounds results.
Two versions which use the interpreter and compiler for running automata; the compiler generates longer-lived objects and more GC work. Both use static load balancing and fixed RNGs for reproducibility.

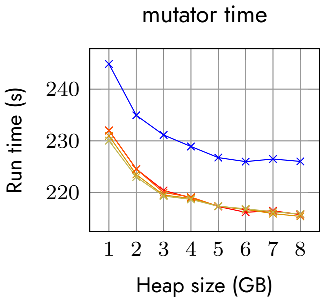


Regrind, interpreted

Parallel mark-region collection works well in small heaps, but is beat by copying in large heaps. Not moving produces better mutator performance than copying for Regrind.
(The mutator times were closer together on my desktop; it might be more hardware-sensitive.)



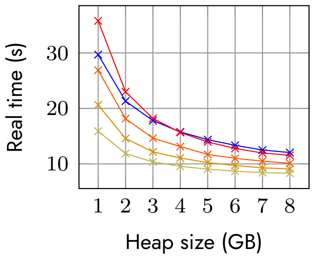




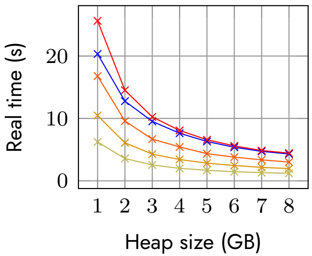
Regrind, compiled

Parallel collection drastically reduces collection time in compiled Regrind. The mutator grabs the allocation lock more frequently in larger heaps using mark-region collection, due to having more densely used pages after a collection.

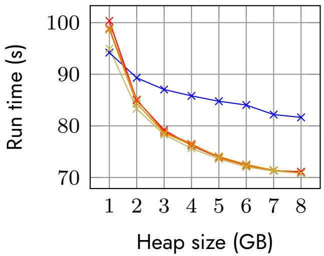
real time



GC time



mutator time



To do

- The heap should be compacted infrequently. I’ve started implementing an *incremental* compaction algorithm, which moves only the most fragmented pages in one collection.
- SBCL uses a separate *immobile space* to store some objects using 32-bit pointers, which we need to implement support for.
- Tracing could be made concurrent, as Ossia et al also have described.

To investigate

- SBCL uses six generations, whereas most garbage collectors just use two. Having fewer would make compaction somewhat easier.
- It may be worthwhile to use *reference counting* for the old generations, which face fewer updates. Reference counting lends itself to an embarrassingly parallel implementation, whereas tracing doesn't, and does work proportional to updates and not objects.
- *Thread-local* garbage collection schemes may help scalability, may make better use of cache memory, and avoid global *stop-the-world* pauses. More important today with more cores, larger caches, quasi-NUMA due to chiplets, and (relatively) slower main memory.

Credit where it's due

- Stas Boukarev and Douglas Katzman helped me get up to speed with how the garbage collector interacts with the rest of SBCL.
- Steve Blackburn and Kunal Sareen discussed Immix with me.
- Nicolas Hafner helped with auto-vectorisation, and guided me around the Kandria source code.
- Paul Khuong and Larry Masinter contributed to the design of the internal memory manager used by the garbage collector.
- Cliff Click helped make sense of the benchmarking results.
- Grindwork Corporation provided me with access to a virtual machine for benchmarking.
- Jan Moringen, Kunal Sareen, Elijah Stone, and Robert Strandh provided comments on early versions of the paper.

Thanks!

(Get the collector from `https://github.com/no-defun-allowed/swcl`, or slides from `https://applied-langua.ge/~hayley/swcl-presentation.pdf`)