Plug & Play Garbage Collection with MMTk

RubyKaigi 2023
Matt Valentine-House (eightbitraptor).

Senior Developer, Ruby & Rails Infrastructure Team, Shopify.

CRuby Committer since March 2023.
1995

Ruby 0.95 released
562
562 \rightarrow 14100
typedef struct RVALUE {
    union {
        struct {
            UINT flag; /* always 0 for freed obj */
            struct RVALUE *next;
        } free;
        // enumerate Ruby object types
    } as;
} RVALUE;
typedef struct RVALUE {
    union {
        struct {
            UINT flag; /* always 0 for freed obj */
            struct RVALUE *next;
        } free;
        // enumerate Ruby object types
        } as;
    } RVALUE;
RVALUE *freelist
“Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1”

John McCarthy
Massachusetts Institute of Technology

gc_mark(obj)
    register RVALUE *obj;
{
    Top:
    if (obj == Qnil) return;    /* nil not marked */
    if (FIXNUM_P(obj)) return;  /* fixnum not marked */
    if (obj->as.basic.flags == 0) return; /* free cell */
    if (obj->as.basic.flags & FL_MARK) return; /* marked */
    obj->as.basic.flags |= FL_MARK;
    switch (obj->as.basic.flags & T_MASK) {
        case T_NIL:
        case T_FIXNUM:
            Bug("gc_mark() called for broken object");
            break;
        /// ... snip...
    }
gc_sweep()
{
    // ... snip
    while (p < pend) {
        if (!(p->as.basic.flags & FL_MARK)) {
            if (p->as.basic.flags) obj_free(p);
            p->as.free.flag = 0;
            p->as.free.next = nfreelist;
            nfreelist = p;
            n++;
        }
        else
            RBASIC(p)->flags &= ~FL_MARK;
        p++;
    }
    // ... snip
}
```c
void gc_sweep()
{
    // ... snip
    while (p < pend) {
        if (!((p->as.basic.flags & FL_MARK))) {
            if (p->as.basic.flags) obj_free(p);
            p->as.free.flag = 0;
            p->as.free.next = nfreelist;
            nfreelist = p;
            n++;
        } else
            RBASIC(p)->flags &= ~FL_MARK;
        p++;
    }
    // ... snip
}
```
2011

Ruby 1.9.3 introduced Lazy Sweeping
Time

Key:  Mutator  Garbage Collection
😊 Lower p99 response times.
Fewer slow requests for users.

😞 Lower throughput.
Fewer total requests per second.
2013

Ruby 2.1 introduced Generational GC
Measurement of Object lifetimes proved that young objects die young and old objects continue to live.

– David Ungar
University of California, 1984
RGenGC - 2 generations: Young and Old

Two phase Marking:

Minor - only young objects

Major - all objects

Minor mark by default, Major when Old object count doubles
RGenGC - 2 generations: Young and Old

Two phase Marking:

Minor - only young objects

Major - all objects

Minor mark by default, Major when Old object count doubles
RGenGC - 2 generations: Young and Old

Two phase Marking:

  Minor - only young objects

  Major - all objects

**Minor mark by default, Major when Old object count doubles**
Young

1

2

3

4

5

6

Old

7

8

9

10

11

12

Reference added from old object to young.
RGenGC introduced Write Barriers.

C extension objects are "Write Barrier Unprotected".

Unprotected objects can never be old.

RGenGC backwards compatible, at expense of performance.
RGenGC introduced Write Barriers.

C extension objects are "Write Barrier Unprotected".

Unprotected objects can never be old.

RGenGC backwards compatible, at expense of performance.
RGenGC introduced Write Barriers.

C extension objects are "Write Barrier Unprotected".

Unprotected objects can never be old.

RGenGC backwards compatible, at expense of performance.
RGenGC introduced Write Barriers.

C extension objects are "Write Barrier Unprotected".

**Unprotected objects can never be old.**

RGenGC backwards compatible, at expense of performance.
RGenGC introduced Write Barriers.

C extension objects are "Write Barrier Unprotected".

Unprotected objects can never be old.

RGenGC backwards compatible, at expense of performance.
Ruby's Generational GC is not evacuating.

Evacuation allows greater performance tuning.
Young object space

Age: 0

Old object space
Young object space

Old object space

Age: 1
Ruby's Generational GC is not evacuating.

Evacuation allows greater performance tuning.
2014

Ruby 2.2 introduced Incremental Marking
Time

Key:  Mutator  Minor Mark  Major Mark  Sweeping
Reference added to already marked object
Write-Barrier recolours object to trigger re-marking
2019

Ruby 2.7 introduced Compaction
First time Objects can move.

2-Finger compaction, from 1960's LISP.

Fits Ruby's memory layout.
First time Objects can move.

2-Finger compaction, from 1960's LISP.

Fits Ruby's memory layout.
First time Objects can move.

2-Finger compaction, from 1960's LISP.

Fits Ruby's memory layout.
Performance Improvements from:

- Smaller heap.
- Better locality.
- Better Copy-on-write performance.
- Extension object pinned by default, explicit opt-in.
Performance Improvements from:

Smaller heap.

Better locality.

Better Copy-on-write performance.

Extension object pinned by default, explicit opt-in.
Performance Improvements from:

Smaller heap.

Better locality.

Better Copy-on-write performance.

*Extension object pinned by default, explicit opt-in.*
static void cont_compact(void *ptr)
{
    rb_context_t *cont = ptr;

    if (cont->self) {
        cont->self = rb_gc_location(cont->self);
    }
    cont->value = rb_gc_location(cont->value);
    rb_execution_context_update(&cont->saved_ec);
}
2020

Ruby 3.0 introduced Automatic Compaction
Ruby 2.7: Manual compaction, 3.0: Automatic compaction

Empty slots filled when swept.

Objects can be modified during sweeping.
Ruby 2.7: Manual compaction, 3.0: Automatic compaction

Empty slots filled when swept.

Objects can be modified during sweeping.
Ruby 2.7: Manual compaction, 3.0: Automatic compaction

Empty slots filled when swept.

**Objects can be modified during sweeping.**
Class

Class

Class

Class
Assumptions made about GC that no longer held.

Solved with read barriers.

Auto-compaction reduced GC performance.
Assumptions made about GC that no longer held.

**Solved with read barriers.**

Auto-compaction reduced GC performance.
Assumptions made about GC that no longer held.

Solved with read barriers.

**Auto-compaction reduced GC performance.**
2022

Ruby 3.2 introduced Variable Width Allocation
Optimizing Ruby’s Memory Layout

Peter Zhu
Ruby Core Committer
Production Engineer, Shopify

Matt Valentine-House
Senior Developer, Shopify

RubyKaigi Takeout 2021 #rubykaigi
Improved mutator performance.

Improved data locality.

fewer external allocations.
Improved mutator performance.

Improved data locality.

fewer external allocations.
Improved mutator performance.

Improved data locality.

fewer external allocations.
2023?
Ruby ships with an incremental, non-copying generation mark & sweep Garbage collector with optional compaction

—Matt Valentine-House
RubyKaigi 2023
Grown organically over 29 years.

Worked around assumptions, and made its own.

This is fine.
Grown organically over 29 years.

**Worked around assumptions, and made its own.**

This is fine.
Grown organically over 29 years.

Worked around assumptions, and made its own.

This is fine.
The alternative is a well worn path that starts down the easy road of reference counting or conservative GC and ends with a system that has a good compiler but is hamstrung by poor memory performance.

–Stephen Blackburn, 2011
Australian National University
Rubys GC is 30 years old.

Core algorithms are >70 years old.

~6% of the entire Ruby core codebase.

Very hard to change.

Weak abstraction boundaries.
Rubys GC is 30 years old.

Core algorithms are $>70$ years old.

~6% of the entire Ruby core codebase.

Very hard to change.

Weak abstraction boundaries.
Rubys GC is 30 years old.

Core algorithms are >70 years old.

~6% of the entire Ruby core codebase.

Very hard to change.

Weak abstraction boundaries.
Rubys GC is 30 years old.

Core algorithms are >70 years old.

~6% of the entire Ruby core codebase.

**Very hard to change.**

Weak abstraction boundaries.
Rubys GC is 30 years old.

Core algorithms are >70 years old.

~6% of the entire Ruby core codebase.

Very hard to change.

**Weak abstraction boundaries.**
“Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance”

Stephen M. Blackburn  
Australian National University

Kathryn S. McKinley  
The University of Texas at Austin

presented at the ACM SIGPLAN Conference on Programming Language Design and Implementation: PLDI 2008
<table>
<thead>
<tr>
<th>Allocation</th>
<th>Identification</th>
<th>Reclamation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocation</td>
<td>Identification</td>
<td>Reclamation</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Free-List</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocation</td>
<td>Identification</td>
<td>Reclamation</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Free-List</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bump Pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocation</td>
<td>Identification</td>
<td>Reclamation</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Free-List</td>
<td>Tracing (Implicit)</td>
<td></td>
</tr>
<tr>
<td>Bump Pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocation</td>
<td>Identification</td>
<td>Reclamation</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Free-List</td>
<td>Tracing (Implicit)</td>
<td></td>
</tr>
<tr>
<td>Bump Pointer</td>
<td><strong>Reference Counting</strong> (Explicit)</td>
<td></td>
</tr>
<tr>
<td>Allocation</td>
<td>Identification</td>
<td>Reclamation</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Free-List</td>
<td>Tracing (Implicit)</td>
<td>Sweep</td>
</tr>
<tr>
<td>Bump Pointer</td>
<td>Reference Counting (Explicit)</td>
<td></td>
</tr>
<tr>
<td>Allocation</td>
<td>Identification</td>
<td>Reclamation</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Free-List</td>
<td>Tracing (Implicit)</td>
<td>Sweep</td>
</tr>
<tr>
<td>Bump Pointer</td>
<td>Reference Counting (Explicit)</td>
<td>Evacuation</td>
</tr>
<tr>
<td>Allocation</td>
<td>Identification</td>
<td>Reclamation</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Free-List</td>
<td>Tracing (Implicit)</td>
<td>Sweep</td>
</tr>
<tr>
<td>Bump Pointer</td>
<td>Reference Counting (Explicit)</td>
<td>Evacuation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compaction</td>
</tr>
</tbody>
</table>
Allocation | Identification | Reclamation
---|---|---
Free-List | Tracing (Implicit) | Sweep
Bump Pointer | Reference Counting (Explicit) | Evacuation
| | | Compaction

Mark-Compact
Each algorithm has different performance characteristics.

Modern GC's: G1, Shenandoah, ZGC et al. are composed from canonical algorithms.

Combined with generations, concurrency and parallelism to achieve high performance.
Each algorithm has different performance characteristics.

**Modern GC's**: G1, Shenandoah, ZGC et al. are composed from canonical algorithms.

Combined with generations, concurrency and parallelism to achieve high performance.
Each algorithm has different performance characteristics.

Modern GC's: G1, Shenandoah, ZGC et al. are composed from canonical algorithms.

Combined with generations, concurrency and parallelism to achieve high performance.
Immix was a new canonical collector.

It formalised Mark-Region as a category of collectors.

Outperforms existing canonical collectors by 7-25% on average.
Immix was a new canonical collector.

It formalised Mark-Region as a category of collectors.

Outperforms existing canonical collectors by 7-25% on average.
Immix was a new canonical collector.

It formalised Mark-Region as a category of collectors.

Outperforms existing canonical collectors by 7-25% on average.
Inko, and Scala Native

Glasgow Haskell Compiler (GHC).

Rubinius
Inko, and Scala Native

Glasgow Haskell Compiler (GHC).

Rubinius
Inko, and Scala Native

Glasgow Haskell Compiler (GHC).

Rubinius
“Low-Latency, High-Throughput Garbage Collection”

Wenyu Zhao  
Australian National University

Stephen M. Blackburn  
Australian National University

Kathryn S. McKinley  
Google

presented at the ACM SIGPLAN Conference on Programming Language Design and Implementation: PLDI 2022
LXR is a new high-level GC algorithm.

Adds reference counting, and heavy optimisations, to Immix.

Significantly outperforms high-profile production GC's.
LXR is a new high-level GC algorithm.

**Adds reference counting, and heavy optimisations, to Immix.**

Significantly outperforms high-profile production GC's.
LXR is a new high-level GC algorithm.

Adds reference counting, and heavy optimisations, to Immix.

**Significantly outperforms high-profile production GC's**
Geometric Mean of 99.99% Latency, and throughput relative to G1, for 4 collectors in three heap sizes

<table>
<thead>
<tr>
<th>Heap size</th>
<th>G1</th>
<th>LXR</th>
<th>Shen</th>
<th>ZGC</th>
<th>G1</th>
<th>LXR</th>
<th>Shen</th>
<th>ZGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3x</td>
<td>1.00</td>
<td>0.72</td>
<td>1.51</td>
<td>-</td>
<td>1.00</td>
<td>0.97</td>
<td>1.77</td>
<td>-</td>
</tr>
<tr>
<td>2x</td>
<td>1.00</td>
<td>0.92</td>
<td>2.54</td>
<td>-</td>
<td>1.00</td>
<td>0.96</td>
<td>0.96</td>
<td>-</td>
</tr>
<tr>
<td>6x</td>
<td>1.00</td>
<td>0.85</td>
<td>1.41</td>
<td>1.44</td>
<td>1.00</td>
<td>1.01</td>
<td>1.09</td>
<td>1.26</td>
</tr>
</tbody>
</table>
Geometric Mean of 99.99% Latency, and throughput relative to G1, for 4 collectors in three heap sizes

<table>
<thead>
<tr>
<th>Heap size</th>
<th>G1</th>
<th>LXR</th>
<th>Shen</th>
<th>ZGC</th>
<th>Time/G1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>G1</td>
<td>LXR</td>
<td>Shen</td>
<td>ZGC</td>
<td>G1</td>
</tr>
<tr>
<td>1.3x</td>
<td>1.00</td>
<td>0.72</td>
<td>1.51</td>
<td>-</td>
<td>1.00</td>
</tr>
<tr>
<td>2x</td>
<td>1.00</td>
<td>0.92</td>
<td>2.54</td>
<td>-</td>
<td>1.00</td>
</tr>
<tr>
<td>6x</td>
<td>1.00</td>
<td>0.85</td>
<td>1.41</td>
<td>1.44</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Geometric Mean of 99.99% Latency, and throughput relative to G1, for 4 collectors in three heap sizes

<table>
<thead>
<tr>
<th>Heap size</th>
<th>99.99% Latency/G1</th>
<th>Time/G1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>G1</td>
<td>LXR</td>
</tr>
<tr>
<td>1.3x</td>
<td>1.00</td>
<td>0.72</td>
</tr>
<tr>
<td>2x</td>
<td>1.00</td>
<td>0.92</td>
</tr>
<tr>
<td>6x</td>
<td>1.00</td>
<td>0.85</td>
</tr>
</tbody>
</table>
MMTk - Part of JikesRVM, from 2004.

Modular design with clear abstractions.

Rust rewrite in 2017, to be runtime agnostic.
MMTk - Part of JikesRVM, from 2004.

**Modular design with clear abstractions.**

Rust rewrite in 2017, to be runtime agnostic.
MMTk - Part of JikesRVM, from 2004.

Modular design with clear abstractions.

Rust rewrite in 2017, to be runtime agnostic.
<table>
<thead>
<tr>
<th>Current</th>
<th>In-Progress</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenJDK</td>
<td></td>
</tr>
<tr>
<td>V8</td>
<td></td>
</tr>
<tr>
<td>JikesRVM</td>
<td></td>
</tr>
<tr>
<td>Current</td>
<td>In-Progress</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>OpenJDK</td>
<td></td>
</tr>
<tr>
<td>V8</td>
<td></td>
</tr>
<tr>
<td>JikesRVM</td>
<td></td>
</tr>
<tr>
<td>Current</td>
<td>In-Progress</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>OpenJDK</td>
<td></td>
</tr>
<tr>
<td>V8</td>
<td></td>
</tr>
<tr>
<td>JikesRVM</td>
<td></td>
</tr>
<tr>
<td>Current</td>
<td>In-Progress</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>OpenJDK</td>
<td>Julia</td>
</tr>
<tr>
<td>V8</td>
<td></td>
</tr>
<tr>
<td>JikesRVM</td>
<td></td>
</tr>
<tr>
<td>Current</td>
<td>In-Progress</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>OpenJDK</td>
<td>Julia</td>
</tr>
<tr>
<td>V8</td>
<td>GHC</td>
</tr>
<tr>
<td>JikesRVM</td>
<td></td>
</tr>
<tr>
<td>Current</td>
<td>In-Progress</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>OpenJDK</td>
<td>Julia</td>
</tr>
<tr>
<td>V8</td>
<td>GHC</td>
</tr>
<tr>
<td>JikesRVM</td>
<td>Ruby</td>
</tr>
</tbody>
</table>
How Can An Existing Language Implement (semi-)Automatically Sped Up?

Laurence Tratt
https://tratt.net/laurie/

2022-04-21
MMTk Core

mmtk-ruby

Language VM
Not production ready.

Linux only, Mac soon, Windows unknown.

MarkSweep runs Rails, Immix does not.

Performance.
Not production ready.

**Linux only, Mac soon, Windows unknown.**

MarkSweep runs Rails, Immix does not.

Performance.
Not production ready.

Linux only, Mac soon, Windows unknown.

**MarkSweep runs Rails, Immix does not.**

Performance.
Not production ready.

Linux only, Mac soon, Windows unknown.

MarkSweep runs Rails, Immix does not.

**Performance.**
Where do we go from here?
#if USE_MMTK
    if (rb_mmtk_enabled_p()) {
        // When using MMTk, we pass the observed
        // ID directly as the `obj` parameter.
        saved.objid = obj;
    } else {
    #endif
        saved.objid = rb_obj_id(obj);
    #if USE_MMTK
    }
    #endif

Complex, error-prone code
Complex, error-prone code makes current Ruby GC harder to change.
Allowed us to get up and running quickly

Complex, error-prone code

Makes current Ruby GC harder to change.
<table>
<thead>
<tr>
<th>Positives</th>
<th>Negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed us to get up and running quickly</td>
<td>Complex, error-prone code</td>
</tr>
<tr>
<td><strong>Helped us to discover areas that we need to change</strong></td>
<td>Makes current Ruby GC harder to change.</td>
</tr>
</tbody>
</table>
V8 & GHC built a GC Interface.

Same interface used by internal GC & MMTk.

We are facing this choice currently in Ruby.

What if we didn't stop with MMTk?
V8 & GHC built a GC Interface.

**Same interface used by internal GC & MMTk.**

We are facing this choice currently in Ruby.

What if we didn't stop with MMTk?
V8 & GHC built a GC Interface.

Same interface used by internal GC & MMTk.

**We are facing this choice currently in Ruby.**

What if we didn't stop with MMTk?
V8 & GHC built a GC Interface.

Same interface used by internal GC & MMTk.

We are facing this choice currently in Ruby.

What if we didn't stop with MMTk?
Can we build a generic memory management interface for Ruby?
<table>
<thead>
<tr>
<th>Benefits</th>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC library</td>
<td></td>
</tr>
<tr>
<td>Benefits</td>
<td>Questions</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>GC library</td>
<td></td>
</tr>
<tr>
<td>MMTk - Latest GC research</td>
<td></td>
</tr>
<tr>
<td>Benefits</td>
<td>Questions</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>GC library</td>
<td></td>
</tr>
<tr>
<td>MMTk - Latest GC research</td>
<td></td>
</tr>
<tr>
<td>Easy GC split testing</td>
<td></td>
</tr>
<tr>
<td>Benefits</td>
<td>Questions</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>GC library</td>
<td></td>
</tr>
<tr>
<td>MMTk - Latest GC research</td>
<td></td>
</tr>
<tr>
<td>Easy GC split testing</td>
<td></td>
</tr>
<tr>
<td>Benefits</td>
<td>Questions</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>GC library</td>
<td>C extensions</td>
</tr>
<tr>
<td>MMTk - Latest GC research</td>
<td></td>
</tr>
<tr>
<td>Easy GC split testing</td>
<td></td>
</tr>
</tbody>
</table>
Add rb_gc_mark_and_move and implement on iseq

#7140

Merged

peterzhu2118 merged 2 commits into ruby:master from Shopify:pz-rb-gc-mark-and-move on Jan 19
[Feature #19406] Allow declarative definition of references for rb_typed_data_struct #7153

eightbitraptor merged 5 commits into ruby:master from Shopify:mvh-declarative-marking on Mar 17
**Benefits**

- GC library
- MMTk - Latest GC research
- Easy GC split testing

**Questions**

- C extensions
- Maintenance burden
<table>
<thead>
<tr>
<th>Benefits</th>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC library</td>
<td>C extensions</td>
</tr>
<tr>
<td>MMTk - Latest GC research</td>
<td>Maintenance burden</td>
</tr>
<tr>
<td>Easy GC split testing</td>
<td>Performance penalties</td>
</tr>
</tbody>
</table>
GC is not a solved problem. Advances in memory management research are happening all the time.

We have a real opportunity to ensure that Ruby's memory management modern and highly performant.
Thanks.

References & Acknowledgements:

bit.ly/mmtk-rubykaigi-2023