An Efficient Non-Moving Garbage Collector for Functional Languages

Katsuhiro Ueno
Atsushi Ohori
Toshiaki Otomo

RIEC, Tohoku University

ICFP 2011,
20 May 2011

Copying Collectors

Mark-and-Sweep Collectors

Copying Collectors

Mark-and-Sweep Collectors

Copying Collectors

Good!

fast allocation
by "bump pointer"

Mark-and-Sweep Collectors

Copying Collectors

Good! fast allocation by "bump pointer"

Good! $O(numLives)$ collection cost

Mark-and-Sweep Collectors

Copying Collectors

Good!

fast allocation by "bump pointer"

Good!

$O(numLives)$ collection cost

many short-lived objects

frequent allocations and collections

Functional Programs

Mark-and-Sweep Collectors

Copying Collectors

Good!

fast allocation
by "bump pointer"

Good!

$O(numLives)$
collection cost

object moving

Good!

good for
generational
collection

many short-lived objects
frequent allocations
and collections

Functional Programs

Mark-and-Sweep Collectors

Bad...

fragmentation
slow allocation

Copying Collectors

Good!

fast allocation
by "bump pointer"

Good!

$O(numLives)$
collection cost

object moving

Good!

good for
generational
collection

many short-lived objects
frequent allocations
and collections

Functional Programs

Mark-and-Sweep Collectors

Bad...
fragmentation
slow allocation

Bad...
$O(heapSize)$
collection cost

Copying Collectors

Good!
fast allocation
by "bump pointer"

Good!
$O(numLives)$
collection cost

object moving

Good!
good for
generational
collection

many short-lived objects
frequent allocations
and collections
Functional Programs

Mark-and-Sweep Collectors

Bad...
fragmentation
slow allocation

Bad...
$O(heapSize)$
collection cost

non-moving

Bad...
no way of
generational
collection

Copying Collectors

Good!
fast allocation
by "bump pointer"

Good!
$O(numLives)$
collection cost

object moving

Good!
good for
generational
collection

many short-lived objects
frequent allocations
and collections
Functional Programs

Mark-and-Sweep Collectors

avoid in practice

Bad...

~~fragmentation~~

~~slow allocation~~

Bad...

$O(heapSize)$ collection cost

non-moving

Bad...

no way of generational collection

Copying Collectors

Good!

fast allocation by "bump pointer"

Good!

$O(numLives)$ collection cost

object moving

Good!

good for generational collection

many short-lived objects

frequent allocations and collections

Functional Programs

Mark-and-Sweep Collectors

avoid in practice

Bad...

~~fragmentation~~

~~slow allocation~~

Bad...

~~$O(heapSize)$~~ $O(numLives)$

collection cost

non-moving

Bad...

no way of generational collection

Copying Collectors

Good!

fast allocation by "bump pointer"

Good!

$O(numLives)$ collection cost

object moving

Good!

good for generational collection

many short-lived objects

frequent allocations and collections

Functional Programs

Mark-and-Sweep Collectors

Bad...

avoid in practice

~~fragmentation~~

~~slow allocation~~

Bad...

~~$O(heapSize)$~~ $O(numLives)$

collection cost

non-moving

↓

Bad...

~~non-moving generational collection~~

non-moving generational collection

any short-lived objects
frequent allocations
and collections

Functional Programs

Copying Collectors

Good!

fast allocation
by "bump pointer"

Good!

$O(numLives)$
collection cost

object moving

↓

Good!

good for
generational
collection

Mark-and-Sweep Collectors

avoid in practice

Bad...

~~fragmentation~~

~~slow allocation~~

Bad...

~~$O(heapSi$~~ $O(numLives)$

collection cost

non-moving

Bad...

non-moving generational collection

any short-lived objects
frequent allocations
and collections

Functional Programs

Copying Collectors

Good!

fast allocation
by "bump pointer"

Good!

$O(numLives)$
collection cost

object moving

Good!

good for
generational
collection

Mark-and-Sweep Collectors

Bad...

~~fragmentation~~

~~slow allocation~~

avoid in practice

Bad...

$O(numLives)$

collection cost

non-moving

Bad...

non-moving
generational
collection

any short-lived objects
frequent allocations
and collections

Functional Programs

Copying Collectors

Good!

fast allocation
by "bump pointer"

Good!

fragmentation-free
by definition

collection cost

object moving

Good!

good for
generational
collection

Mark-and-Sweep Collectors

avoid in practice

Bad...

~~fragmentation~~

~~slow allocation~~

Bad...

~~$O(heapSi$~~ $O(numLives)$

collection cost

non-moving

↓

Bad...

~~no~~

~~ge~~

~~co~~

non-moving generational collection

Copying Collectors

Good!

fast allocation by "bump pointer"

Good!

fragmentation-free by definition

ves)

collection cost

Good!

keep good locality

object moving

↓

Good!

good for generational collection

any short-lived objects frequent allocations and collections

Functional Programs

Mark-and-Sweep Collectors

avoid in practice

Bad...

~~fragmentation~~

~~slow allocation~~

Bad...

~~$O(heapSi$~~ $O(numLives)$

collection cost

non-moving

Bad...

non-moving

generational

collection

Copying Collectors

Good!

fast allocation

by "bump pointer"

Good!

fragmentation-free

by definition

$ves)$

collection cost

Good!

keep good locality

object moving

Good!

good for

generational

collection

any short-lived objects

frequent allocations

and collections

Functional Programs

Mark-and-Sweep Collectors

Copying Collectors

avoid in practice

Bad!

non-moving itself is also a good property!

Bad!

collection cost

Good!

fast allocation by "bump pointer"

Good!

fragmentation-free by definition

collection cost

Good!

keep good locality

non-moving

object moving

Bad!

non-moving generational collection

any short-lived objects frequent allocations and collections

Functional Programs

Good!

good for generational collection

Viva! Non-moving

Very easy to share heap objects.

- No need to do *pinning* when interacting with C and other languages.

Very easy to maximize concurrency.

- No need to stop threads for managing shared objects.

These are *free* when you choose non-moving!

Mark-and-Sweep Collectors

avoid in practice

Bad...

~~fragmentation~~

~~slow allocation~~

Bad...

~~$O(heapSi$~~ $O(numLives)$

collection cost

non-moving

Bad...

non-moving
generational
collection

any short-lived objects
frequent allocations
and collections

Functional Programs

Copying Collectors

Good!

fast allocation
by "bump pointer"

Good!

$O(numLives)$
collection cost

object moving

Good!

good for
generational
collection

Mark-and-Sweep Collectors

avoid in practice

Bad...

~~fragmentation~~

~~slow allocation~~

Bad...

~~$O(heapSi$~~ $O(numLives)$

collection cost

non-moving

↓

Bad...

non-moving generational collection

Copying Collectors

Good!

fast allocation by "bump pointer"

Good!

$O(numLives)$ collection cost

object moving

↓

Good!

good for generational collection

any short-lived objects

frequent allocations and collections

Functional Programs

Mark-and-Sweep Collectors

Bad...

~~fragmentation~~

~~slow allocation~~

avoid in practice

Bad...

~~$O(heapSi$~~ $O(numLives)$

collection cost

non-moving

Bad...

non-moving
generational
collection

Copying Collectors

Good!

fast allocation
by "bump pointer"

Good!

$O(numLives)$
collection cost

object moving

Good!

good for
generational
collection

choice

any short-lived objects
frequent allocations
and collections

Functional Programs

The topic of this talk

We propose
a *non-moving* GC
which is as efficient as
Cheney's copying GC.

Weakness of mark-and-sweep GC

- Fragmentation, and slow allocation
- High sweep cost ($O(heapSize)$)
- No known method for extending it to non-moving generational GC

We choose a well-known idea of bitmap marking as our start point to overcome these weaknesses...

Strategy

But bitmap marking strategy alone does not yield an efficient GC.

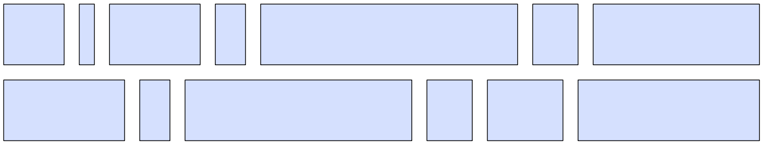
We re-organize the bitmap marking with

- fragmentation avoiding heap organization
- tree structured bitmap
- automatic heap size adjustment
- non-moving generational extension

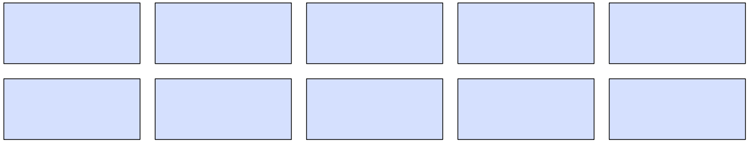
with a series of optimized bit manipulation algorithms.

Fragmentation

Varied size objects incur fragmentation.

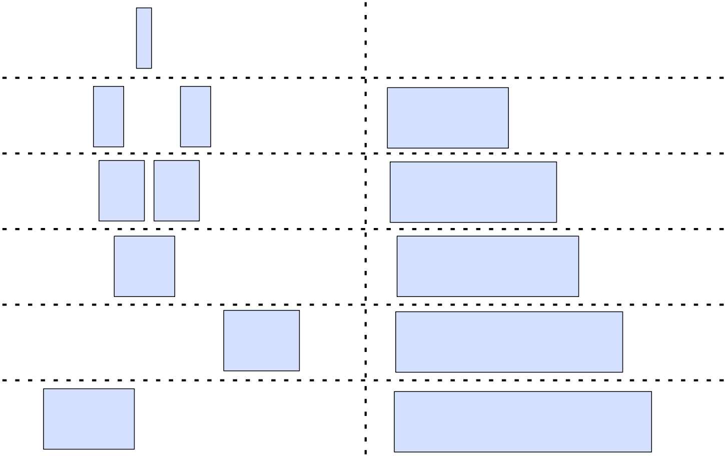


If all objects were same size, heap could be a fixed size array without incurring fragmentation.



Avoid fragmentation

Separate the heap for each object size.



Avoid fragmentation

Separate the heap for each object size.

H_3

8 bytes/block

H_4

16 bytes/block

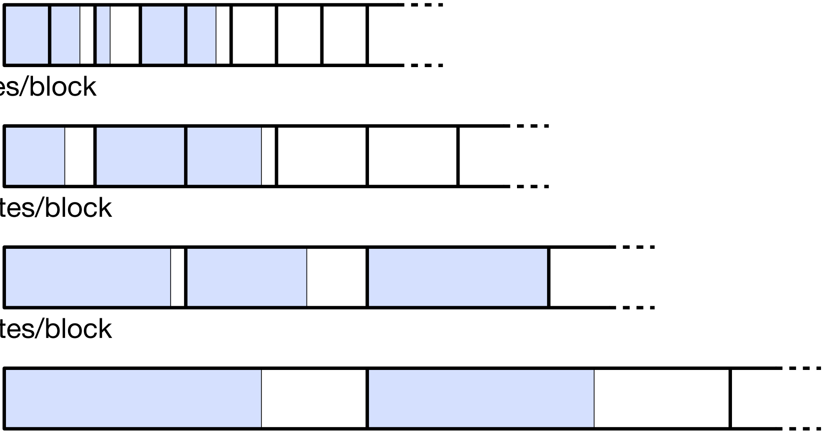
H_5

32 bytes/block

H_6

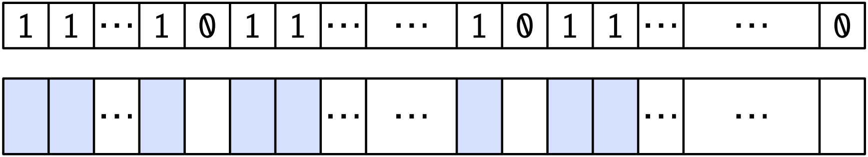
64 bytes/block

⋮



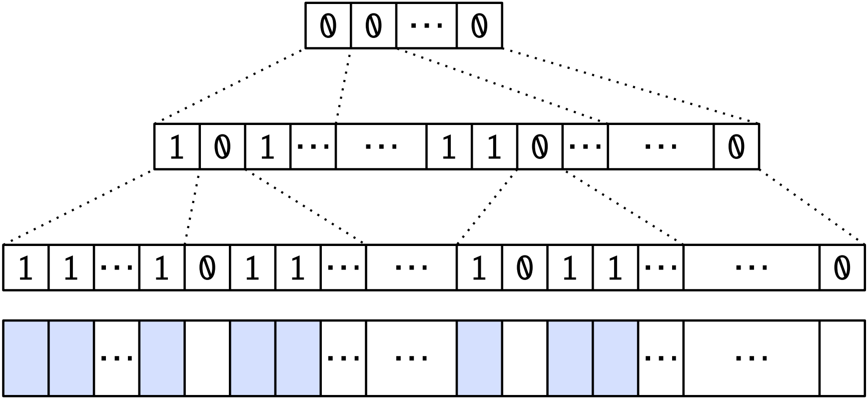
Tree-structured bitmaps

For fast allocation and collection



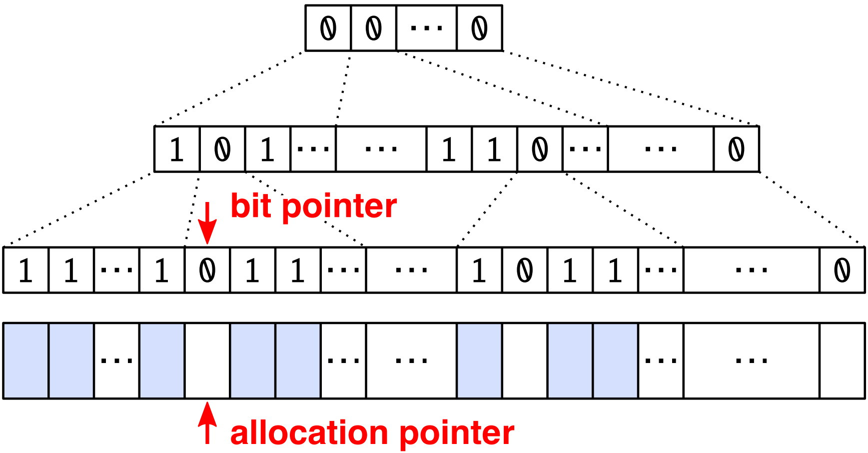
Tree-structured bitmaps

For fast allocation and collection



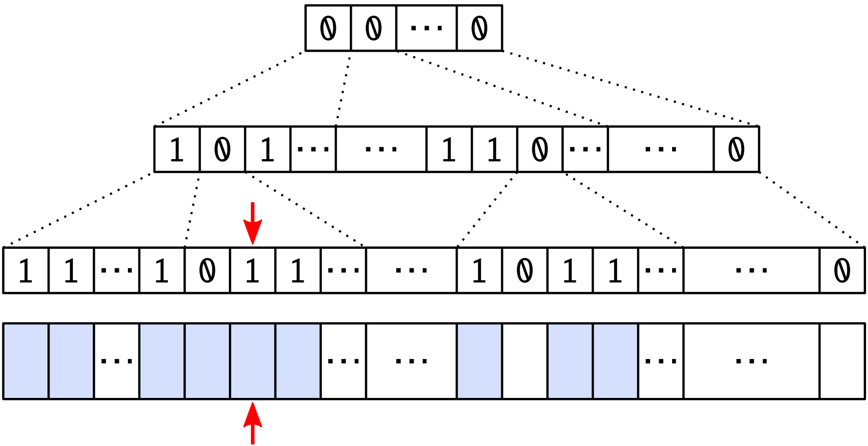
Tree-structured bitmaps

For fast allocation and collection



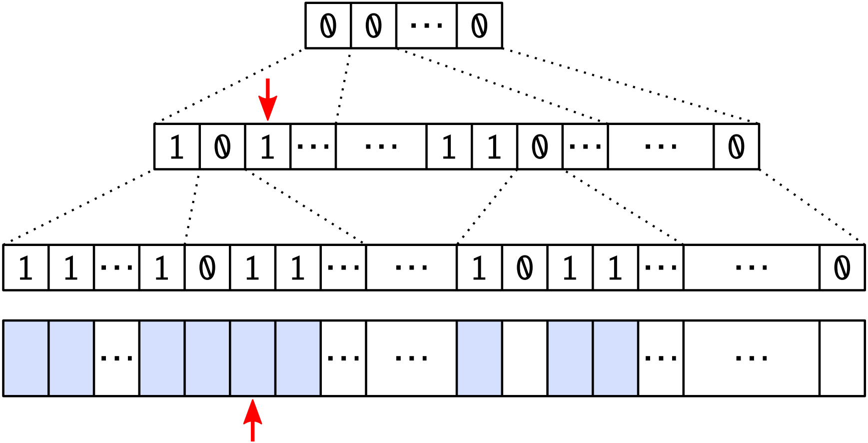
Tree-structured bitmaps

For fast allocation and collection



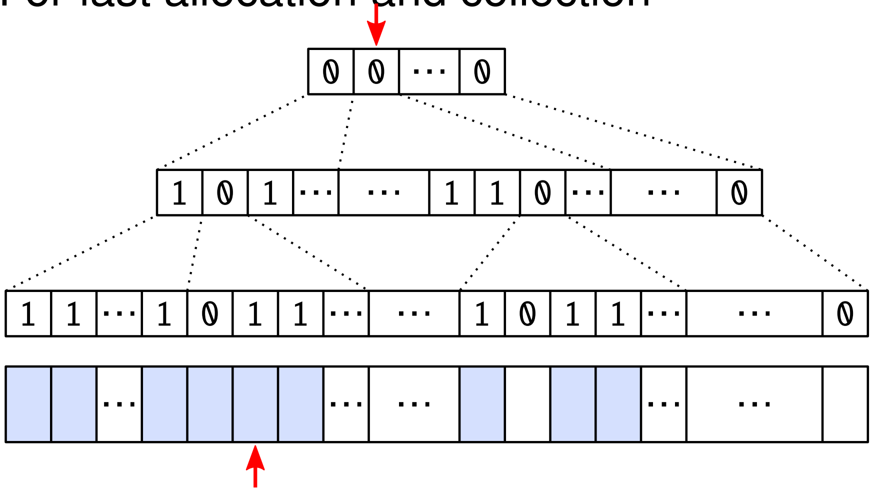
Tree-structured bitmaps

For fast allocation and collection



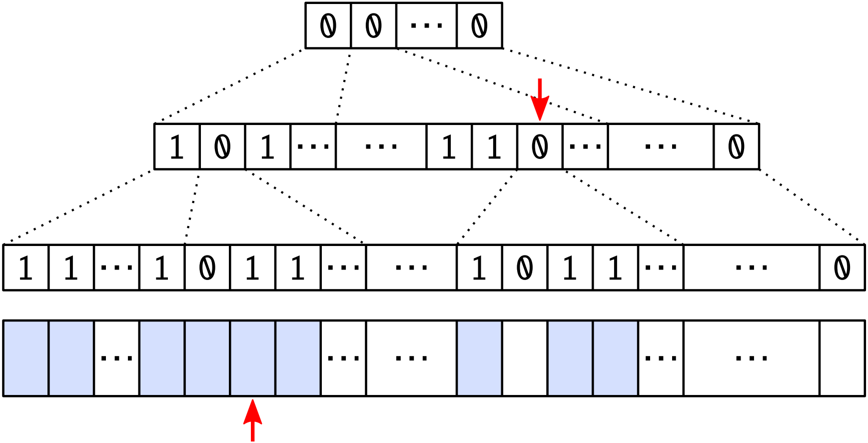
Tree-structured bitmaps

For fast allocation and collection



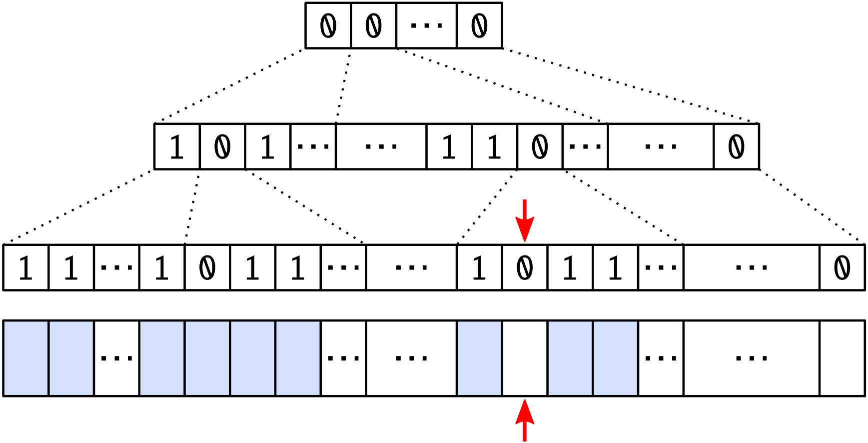
Tree-structured bitmaps

For fast allocation and collection



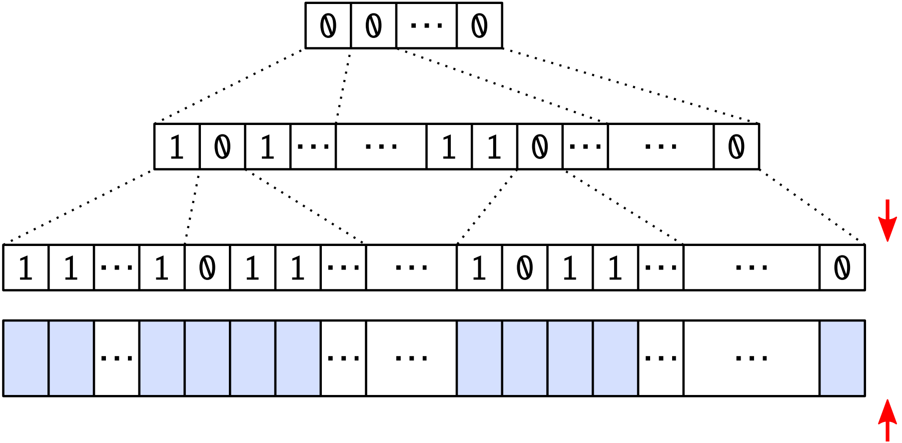
Tree-structured bitmaps

For fast allocation and collection



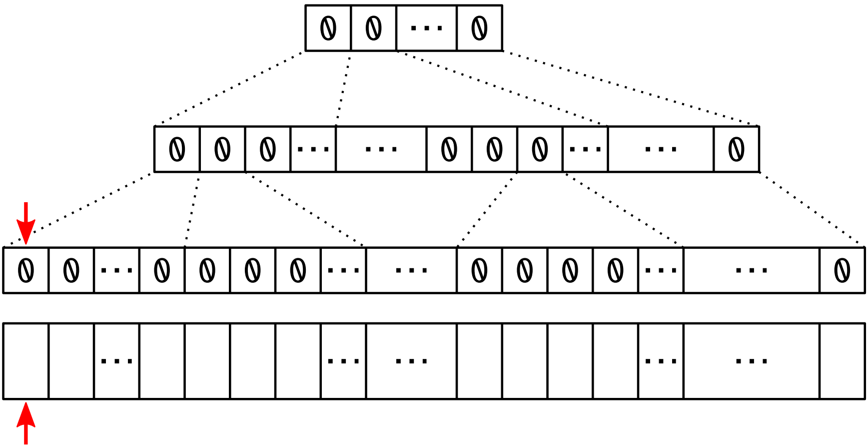
Tree-structured bitmaps

For fast allocation and collection



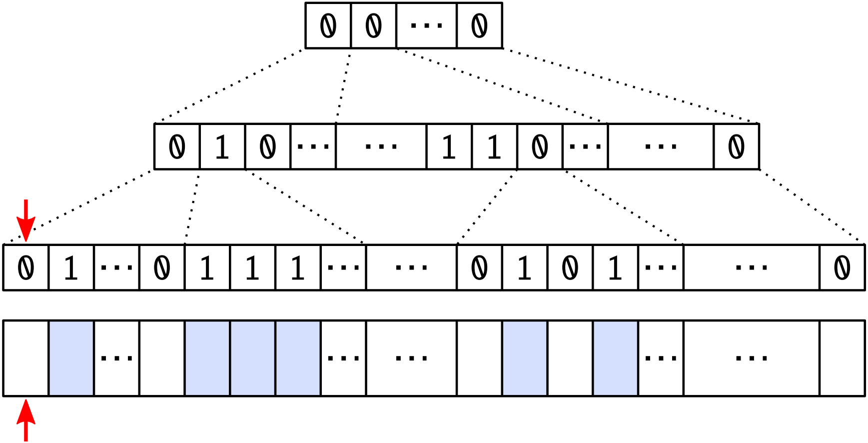
Tree-structured bitmaps

For fast allocation and collection

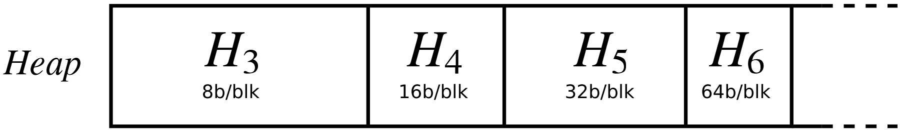


Tree-structured bitmaps

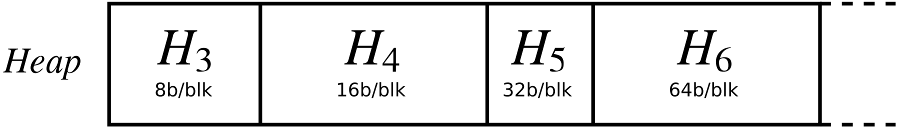
For fast allocation and collection



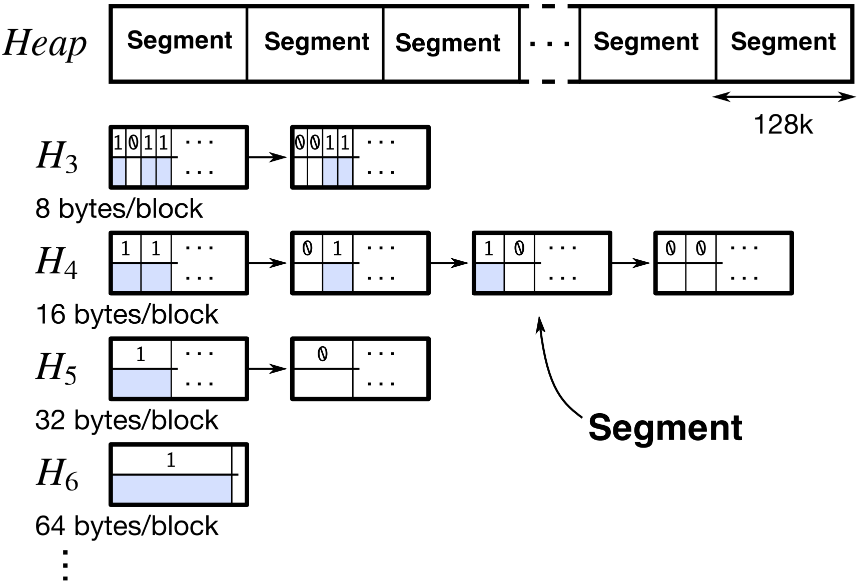
Sub-heap size adjustment



Which layout is appropriate?
... cannot determine it in advance.



Sub-heap size adjustment



Generational extension

Refining the idea of partial GC (Demers et al 1990)

- generations = disjoint sets of objects
- sets of objects = bitmaps

$$\mathcal{G}_3 = \{ \quad O_3, \quad O_4, \quad \ldots \}$$

$$\mathcal{G}_2 = \{ \quad O_2, \quad O_6, \quad \ldots \}$$

$$\mathcal{G}_1 = \{ O_1, \quad O_5, \quad O_7, O_8, \ldots \}$$

block array

O_1	O_2	$\cdots$	O_3		O_4	O_5	$\cdots$	$\cdots$	O_6		O_7	O_8	$\cdots$

Generational extension

Refining the idea of partial GC (Demers et al 1990)

- generations = disjoint sets of objects
- sets of objects = bitmaps

$\mathcal{G}_3 =$	0	0	$\cdots$	1	0	1	0	$\cdots$	$\cdots$	0	0	0	0	$\cdots$
$\mathcal{G}_3 \cup \mathcal{G}_2 =$	0	1	$\cdots$	1	0	1	0	$\cdots$	$\cdots$	1	0	0	0	$\cdots$
$\mathcal{G}_3 \cup \mathcal{G}_2 \cup \mathcal{G}_1 =$	1	1	$\cdots$	1	0	1	1	$\cdots$	$\cdots$	1	0	1	1	$\cdots$
block array	O_1	O_2	$\cdots$	O_3		O_4	O_5	$\cdots$	$\cdots$	O_6		O_7	O_8	$\cdots$

Performance evaluation

We compared

- our method,
- Cheney's copying collector,
- our method with 2 generations, and
- generational copying (based on Reppy 1994)

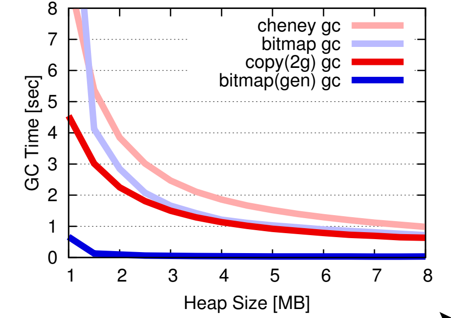
by extensive benchmarks on our SML# compiler.

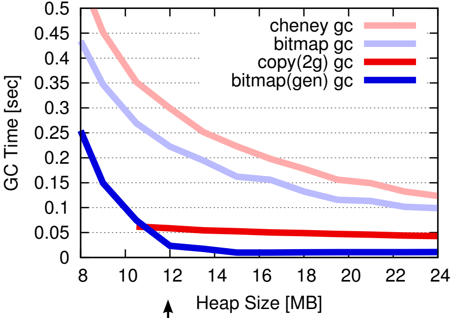
Memory usage

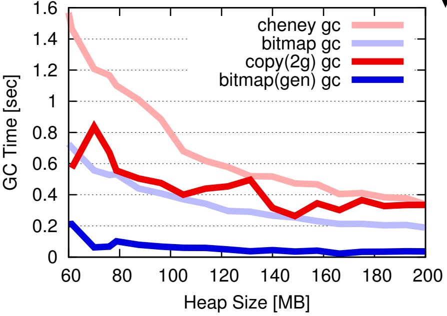
benchmark	size (MB)	our method		copying	
		live	occ.	live	occ.
count_graphs	2	6.02	55.4	6.02	48.7
cpstak	2	5.56	51.6	5.55	48.9
knuth_bendix	12	10.45	61.1	10.17	46.1
ratio_regions	20	11.97	62.3	11.95	47.4
gcbench	65	10.61	65.9	10.25	42.9
perm9	190	22.36	57.6	16.91	41.6

- live : ratio of survivals against GC
- occ. : ratio of memory amount filled with data

Benchmark : GC time





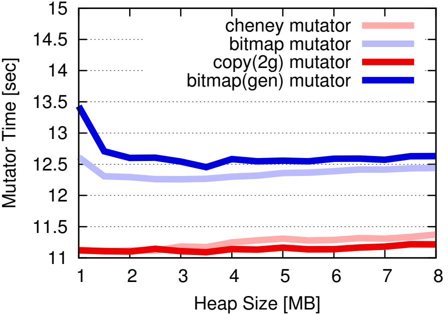


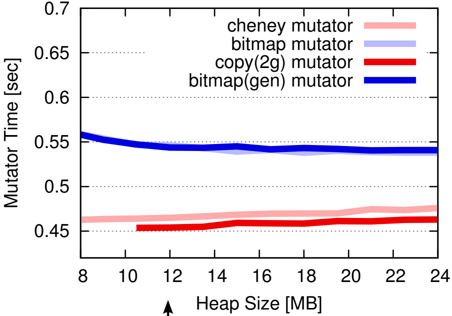
knuth_bendix

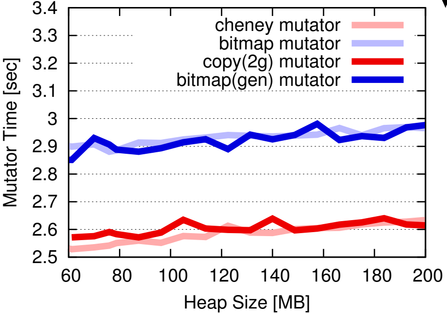
count_graphs

gcbench

Benchmark : mutator time





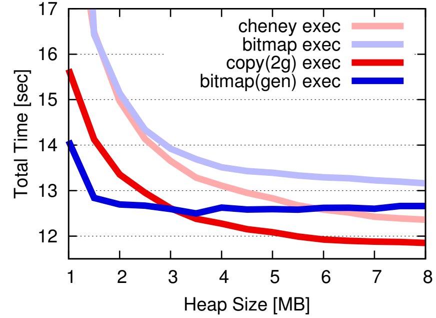


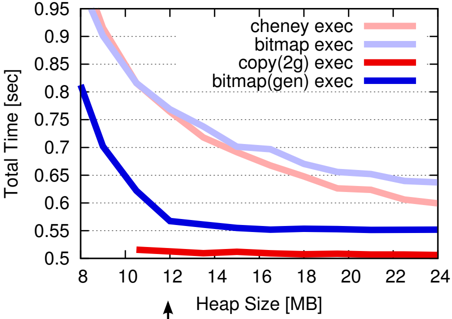
knuth_bendix

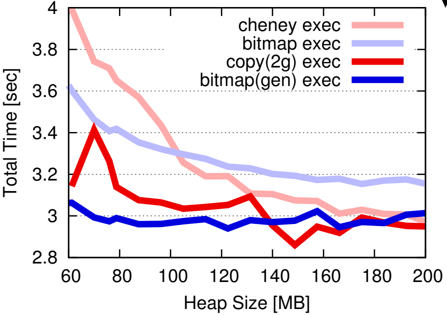
count_graphs

gcbench

Benchmark : total exec time





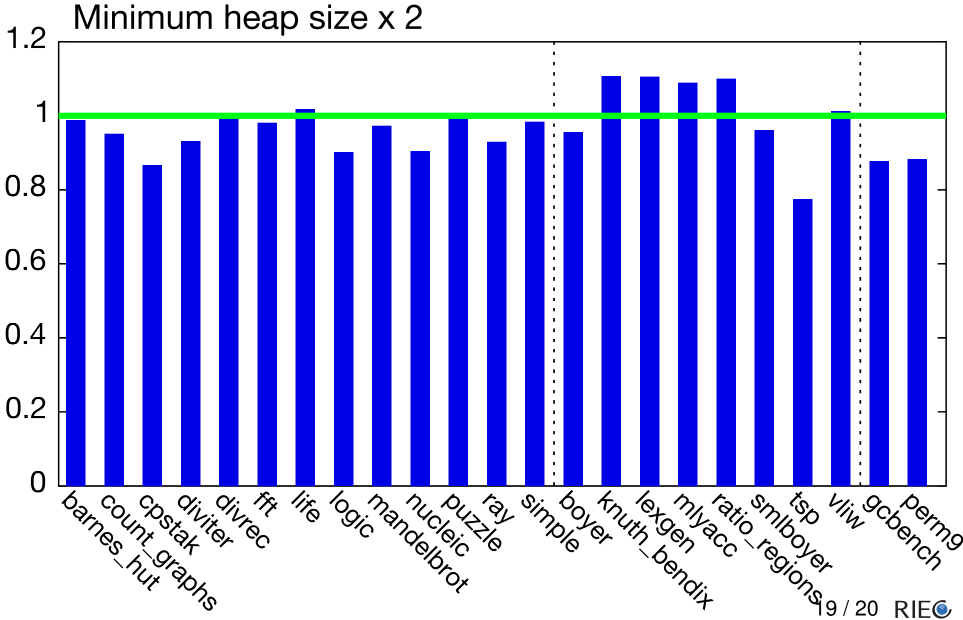


knuth_bendix

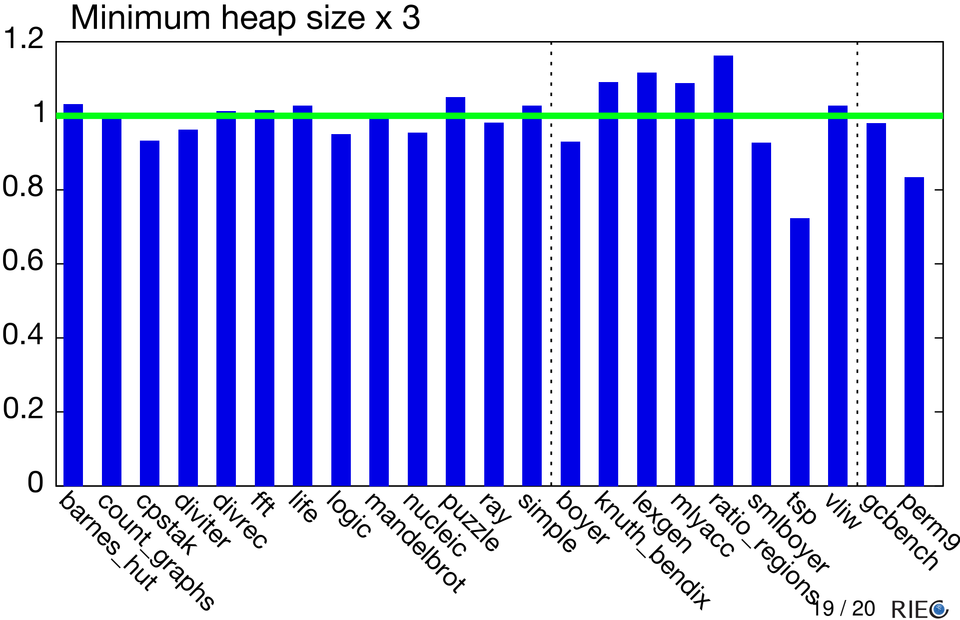
count_graphs

gcbench

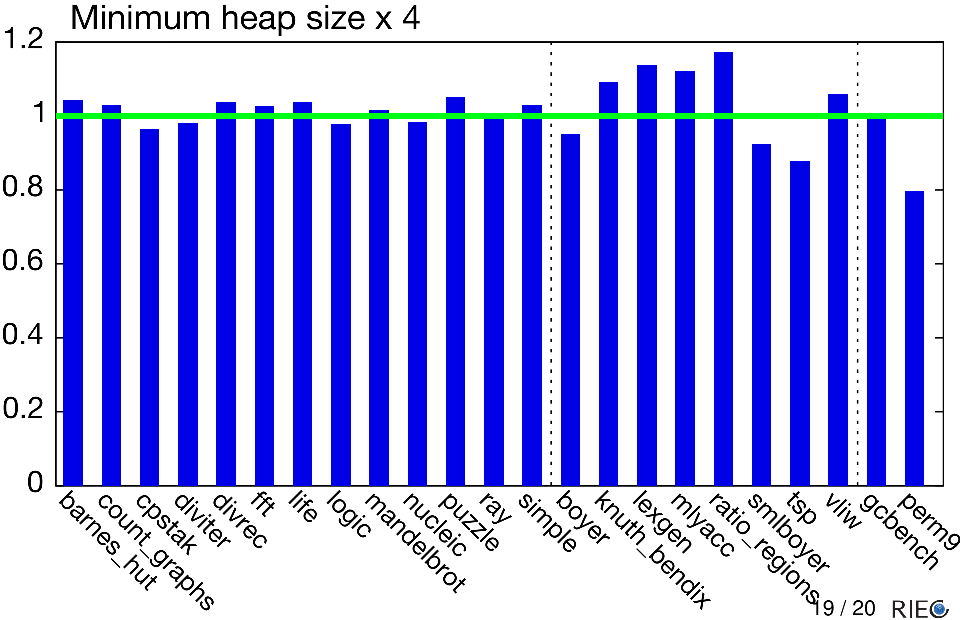
Ratio of total time (ours / copying)



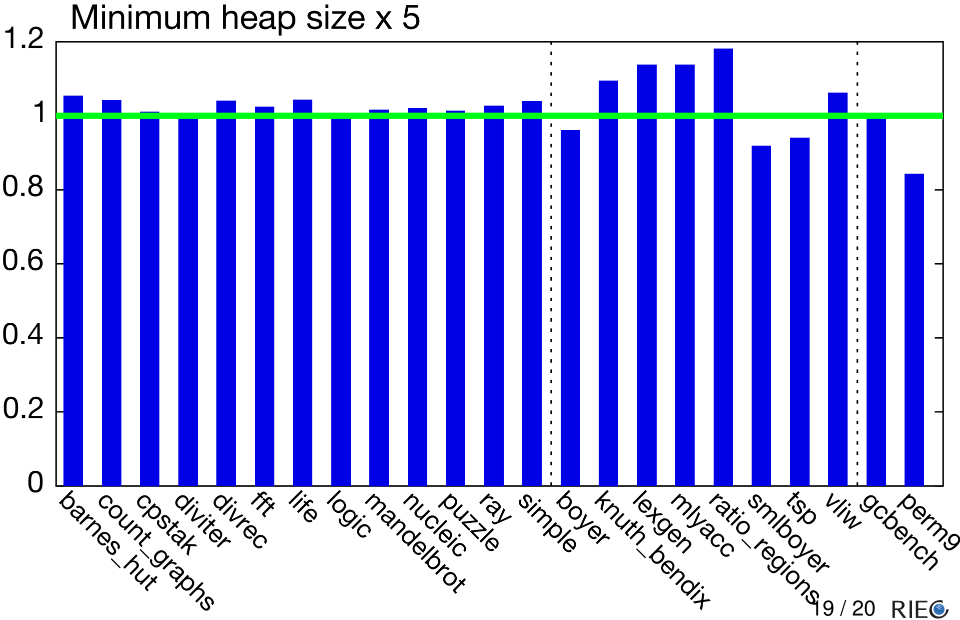
Ratio of total time (ours / copying)



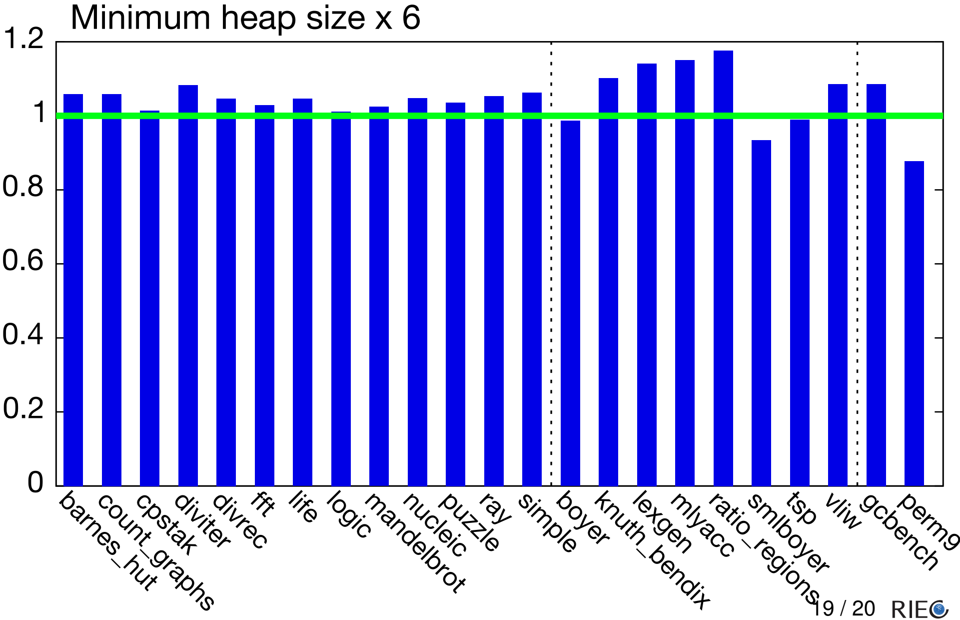
Ratio of total time (ours / copying)



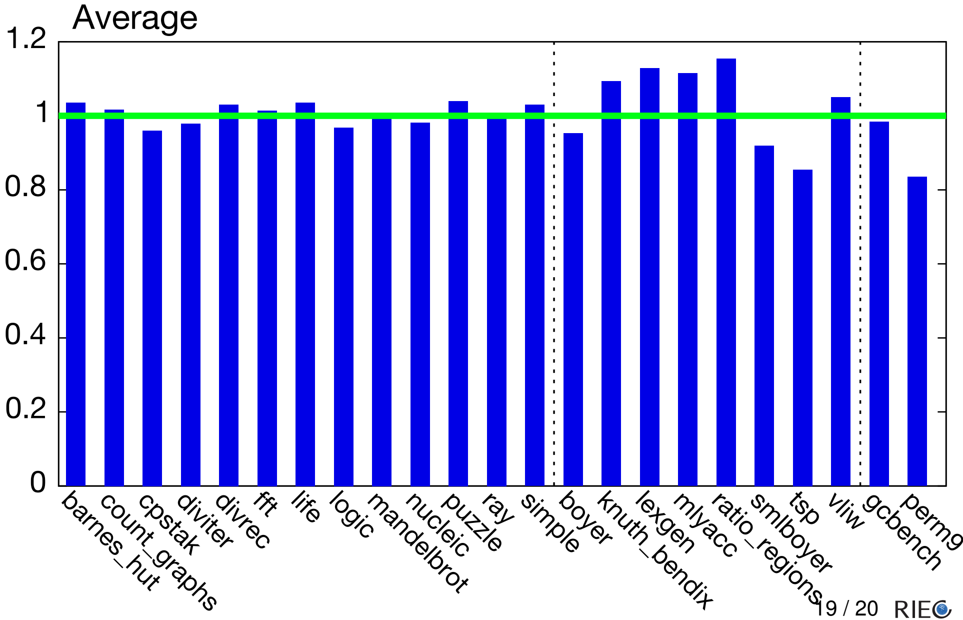
Ratio of total time (ours / copying)



Ratio of total time (ours / copying)



Ratio of total time (ours / copying)



Conclusion

We have developed an efficient non-moving GC.

- avoid fragmentation by separating the heap.
- fast allocation and GC through bitmap trees.
- generational GC through multiple bitmaps.

A viable alternative to copying GC for functional languages.

Further Development

non-moving concurrent GC