Fusuma: Double-Ended Threaded Compaction

Hiro Onozawa[*] Tomoharu Ugawa[**] Hideya Iwasaki[*]

* Graduate School of Informatics and Engineering
The University of Electro-Communications
** Graduate School of Information Science and Technology
The University of Tokyo

22 June 2021

Fusuma: Double-Ended Threaded Compaction

Hiro Onozawa* Tomoharu Ugawa** Hideya Iwasaki*

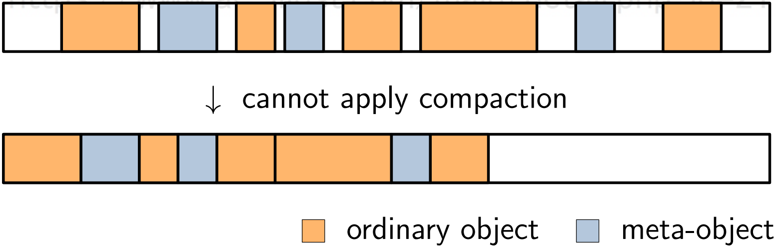
* Graduate School of Informatics and Engineering
The University of Electro-Communications
** Graduate School of Information Science and Technology
The University of Tokyo

22 June 2021

Overview of this paper

In an implementation of a managed language with GC:

- The layout information of ordinary objects may be recorded in meta-objects.
- Existing sliding compaction cannot be applied to a heap where ordinary objects and meta-objects are intermingled.



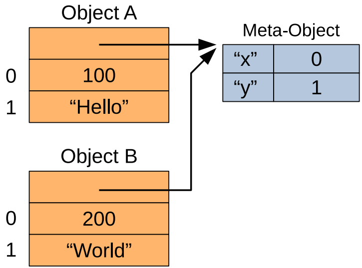
- We propose and evaluate a new compaction algorithm named Fusuma that solves this problem.

Meta-objects

Meta-objects are special objects that have layout information of ordinary objects.

- Java: class objects.
- JavaScript: hidden classes.


```
class C {
  int x;
  String y;

  public C(int x, String y) {
    this.x = x; this.y = y;
  }
}

C A = new C(100, "Hello");
C B = new C(200, "World");
```



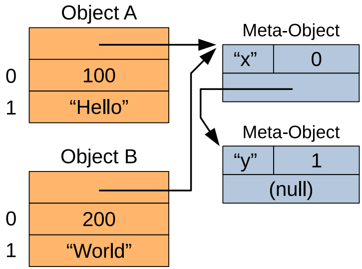
Meta-objects

Meta-objects are special objects that have layout information of ordinary objects.

- Java: class objects.
- JavaScript: hidden classes.

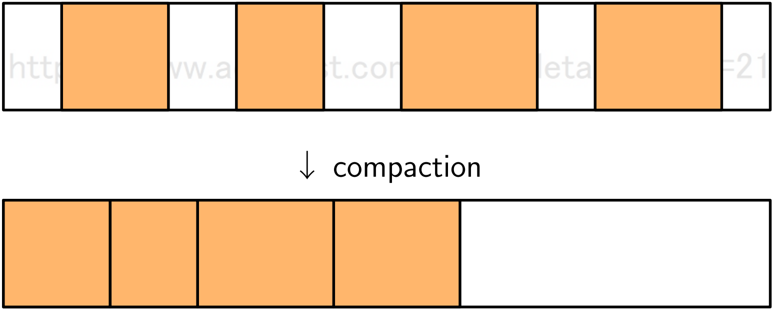
Layout information for an ordinary object may consist of multiple meta-objects.


```
class C {
  int x;
  String y;

  public C(int x, String y) {
    this.x = x; this.y = y;
  }
}

C A = new C(100, "Hello");
C B = new C(200, "World");
```



Sliding compaction GC

Sliding compaction slides live objects to one end of the heap.
⇒ Fragmentation can be eliminated.



Why sliding compaction?

We are developing a JavaScript engine named eJS for IoT devices where approx. 100KB of heap is available.

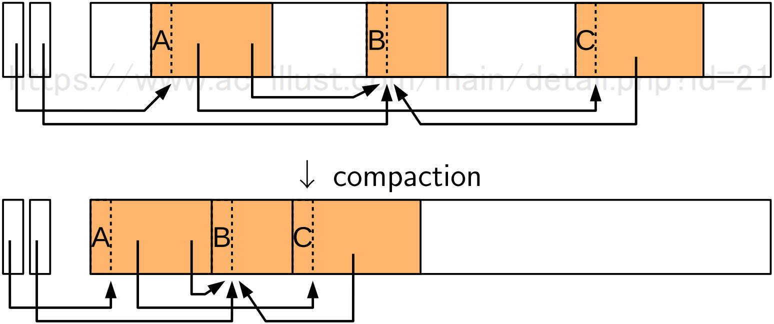
Mark-sweep GC and Copying GC are not space efficient.

⇒ We focused on sliding compaction, especially **Jonkers's threaded compaction** which needs no extra space.

Problems in sliding compaction

Sliding compaction needs to

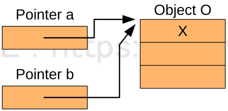
- move every object, and
- update all pointers to objects with their new addresses.



For this reason, sliding compaction needs to know the locations of pointers in an object. ⇒ The layout of each object must be known.

Jonkers's threaded compaction

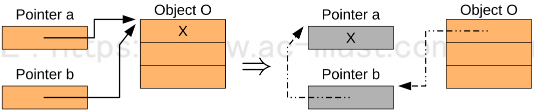
It transforms multiple pointers to the same object into a linked list of pointers' location **without any extra space**.



- Once threading is performed, the pointer cannot be followed until compaction has been completed.

Jonkers's threaded compaction

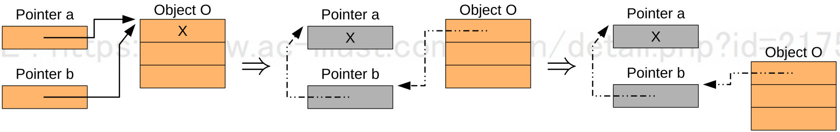
It transforms multiple pointers to the same object into a linked list of pointers' location **without any extra space**.



- Once threading is performed, the pointer cannot be followed until compaction has been completed.

Jonkers's threaded compaction

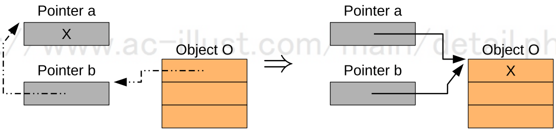
It transforms multiple pointers to the same object into a linked list of pointers' location **without any extra space**.



- Once threading is performed, the pointer cannot be followed until compaction has been completed.

Jonkers's threaded compaction

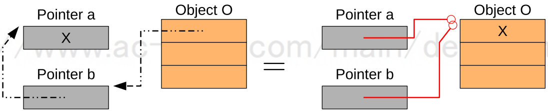
It transforms multiple pointers to the same object into a linked list of pointers' location **without any extra space**.



- Once threading is performed, the pointer cannot be followed until compaction has been completed.

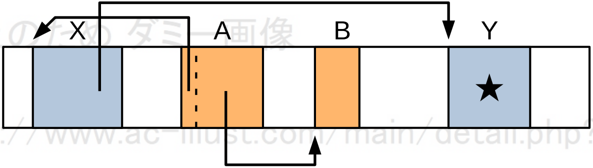
Jonkers's threaded compaction

It transforms multiple pointers to the same object into a linked list of pointers' location **without any extra space**.



- Once threading is performed, the pointer cannot be followed until compaction has been completed.
- In this presentation, we denote threaded pointers in red color.

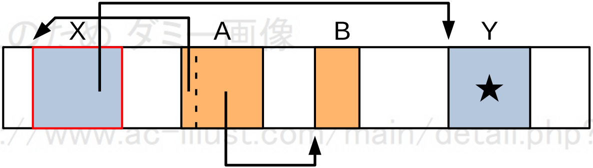
Example of failure in threading



ordinary object   meta-object   ★ layout information

GC must access ★ when threading A.

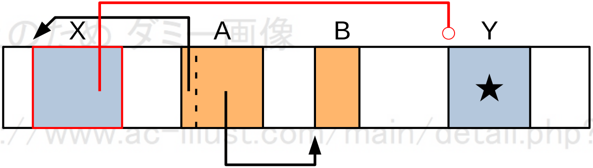
Example of failure in threading



ordinary object   meta-object   ★ layout information

After marking live objects, scans the heap from left to right to search a live object.
⇒ Meta-object X is found.

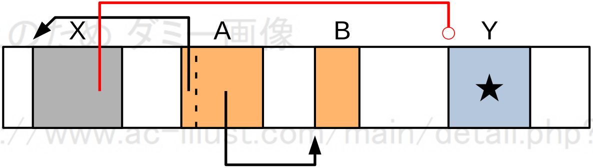
Example of failure in threading



ordinary object　meta-object　★ layout information

Threads the pointer from X to Y.

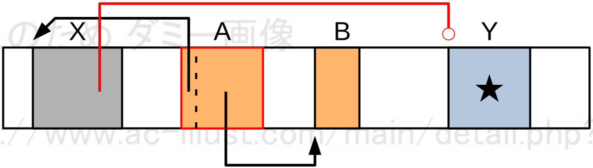
Example of failure in threading



ordinary object   meta-object   ★ layout information

Processing X is over.
Scans the heap for the next live object.

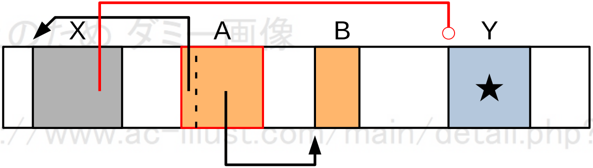
Example of failure in threading



ordinary object   meta-object   ★ layout information

Ordinary object A is found.
To know the location of pointer to B, ★ in meta-object Y is necessary.

Example of failure in threading

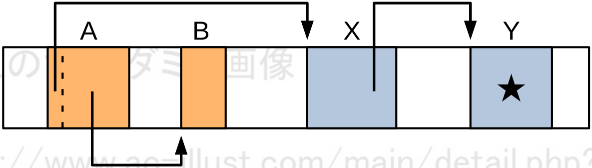


■ ordinary object ■ meta-object ★ layout information

Unfortunately, since pointer to Y in X is threaded, we cannot access ★ in Y and cannot know the pointer location in A.

Example of success in threading

Jonkers's algorithm happens to succeed for the following heap.

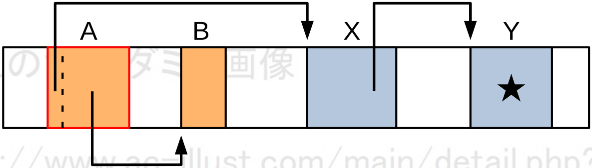


■ ordinary object ■ meta-object ★ layout information

All ordinary objects are to the left of meta-objects.

Example of success in threading

Jonkers's algorithm happens to succeed for the following heap.

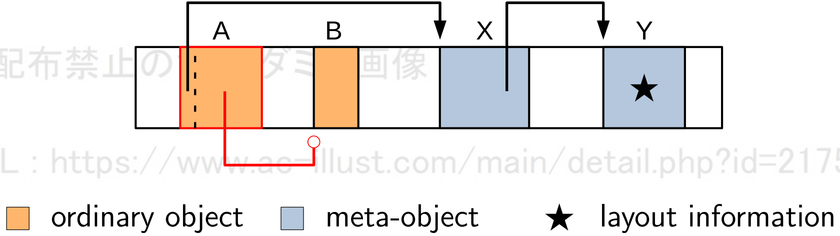


ordinary object   meta-object   ★ layout information

Scans the heap from left to right.
$\Rightarrow$ Ordinary object A is found.

Example of success in threading

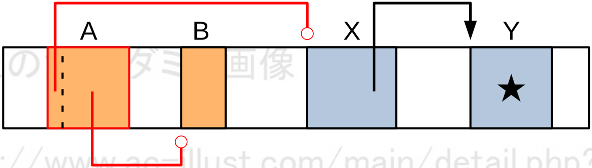
Jonkers's algorithm happens to succeed for the following heap.



Because we have not threaded the pointer to X, we can access ★.
⇒ We can successfully thread the pointer to B.

Example of success in threading

Jonkers's algorithm happens to succeed for the following heap.

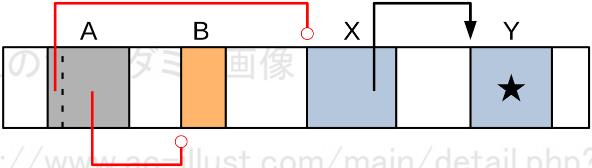


■ ordinary object ■ meta-object ★ layout information

Threads the pointer to meta-object X

Example of success in threading

Jonkers's algorithm happens to succeed for the following heap.

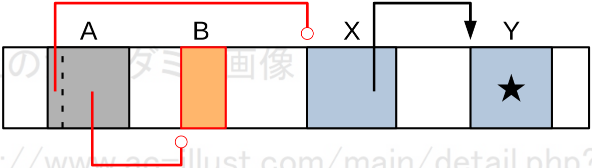


ordinary object　meta-object　★ layout information

Processing A is over.
Goes to the next live object.

Example of success in threading

Jonkers's algorithm happens to succeed for the following heap.

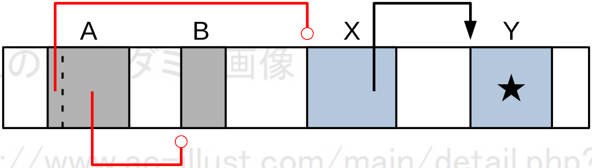


■ ordinary object ■ meta-object ★ layout information

Ordinary object B is found.

Example of success in threading

Jonkers's algorithm happens to succeed for the following heap.

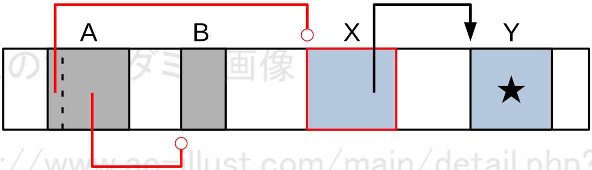


ordinary object meta-object ★ layout information

For B, nothing to do.
Goes to the next live object.

Example of success in threading

Jonkers's algorithm happens to succeed for the following heap.

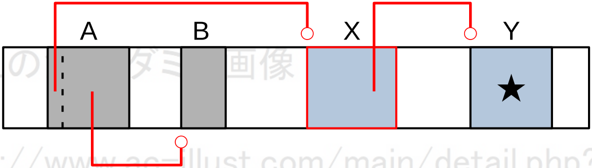


ordinary object   meta-object   ★ layout information

Meta-object X is found.

Example of success in threading

Jonkers's algorithm happens to succeed for the following heap.

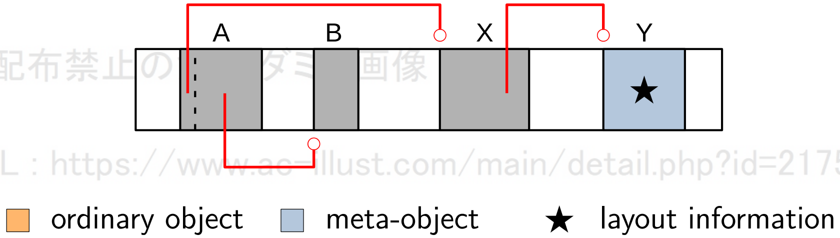


ordinary object   meta-object   ★ layout information

Threads the pointer to Y.

Example of success in threading

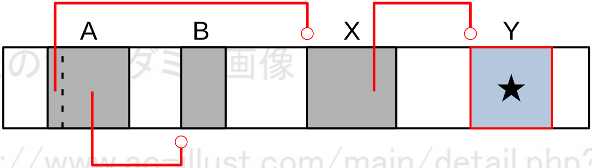
Jonkers's algorithm happens to succeed for the following heap.



Processing X is over.
Goes to the next live object.

Example of success in threading

Jonkers's algorithm happens to succeed for the following heap.

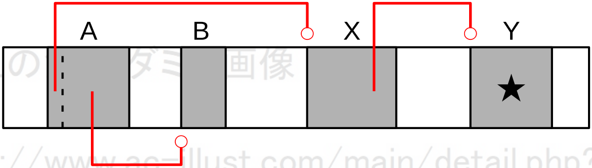


■ ordinary object ■ meta-object ★ layout information

Meta-object Y is found.

Example of success in threading

Jonkers's algorithm happens to succeed for the following heap.

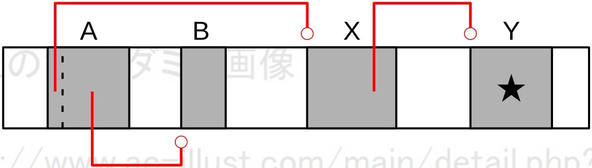


ordinary object | meta-object | ★ layout information

Nothing to do for Y.
We have successfully processed all live objects in the heap.

Example of success in threading

Jonkers's algorithm happens to succeed for the following heap.



ordinary object   meta-object   ★ layout information

The reasons of success are:

- All live ordinary objects are to the left of live meta-objects.
- Consequently, all ordinary objects are processed before any meta-object is processed.

⇒ If we can enforce this processing order, the problem can be solved.

Proposed algorithm: heap observation

Based on this observation, we invented the Fusuma compaction by extending the Jonkers's compaction.

If we process all ordinary objects before we start processing meta-objects, everything should be fine.

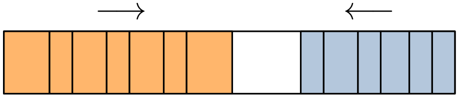
$\Downarrow$

To maintain this processing order, it would be better if ordinary objects and meta-objects were not intermingled.

$\Downarrow$

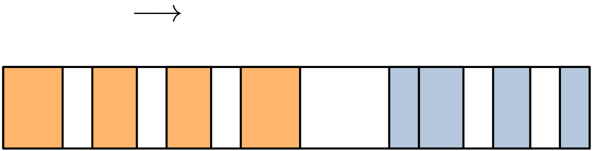
We allocate:

- ordinary objects from the left of the heap in the forward direction
- meta-objects from the right of the heap in the backward direction.



Proposed algorithm: compaction

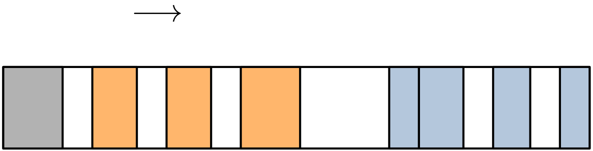
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We scan the ordinary object area **from left to right** and process every live ordinary object.

Proposed algorithm: compaction

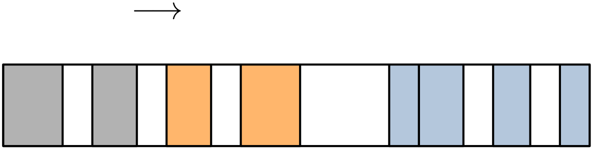
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We scan the ordinary object area **from left to right** and process every live ordinary object.

Proposed algorithm: compaction

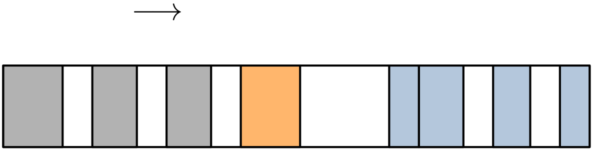
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We scan the ordinary object area **from left to right** and process every live ordinary object.

Proposed algorithm: compaction

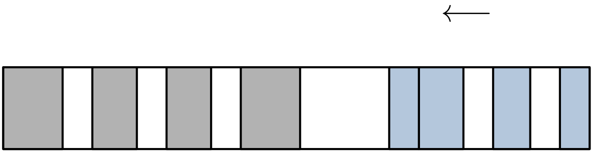
	Ordinary objects	Meta-objects
Allocation	⟶	⟵
Scanning	⟶	⟵
Sliding	⟵	⟶



We scan the ordinary object area **from left to right** and process every live ordinary object.

Proposed algorithm: compaction

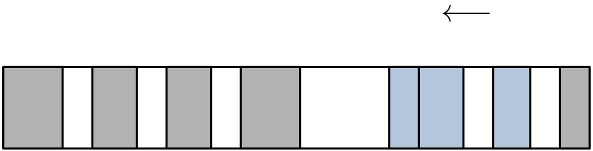
	Ordinary objects	Meta-objects
Allocation	⟶	⟵
Scanning	⟶	⟵
Sliding	⟵	⟶



We scan the meta-object area **from right to left** and process every live meta-object.

Proposed algorithm: compaction

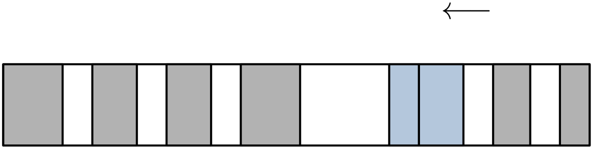
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We scan the meta-object area **from right to left** and process every live meta-object.

Proposed algorithm: compaction

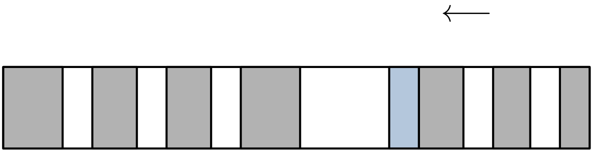
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We scan the meta-object area **from right to left** and process every live meta-object.

Proposed algorithm: compaction

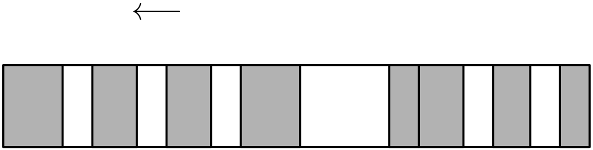
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We scan the meta-object area **from right to left** and process every live meta-object.

Proposed algorithm: compaction

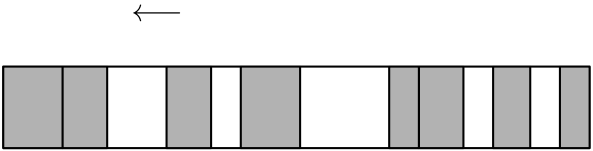
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We slide every live ordinary object **from right to left**.

Proposed algorithm: compaction

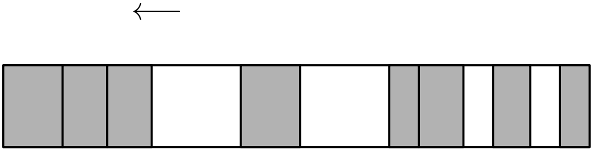
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We slide every live ordinary object **from right to left**.

Proposed algorithm: compaction

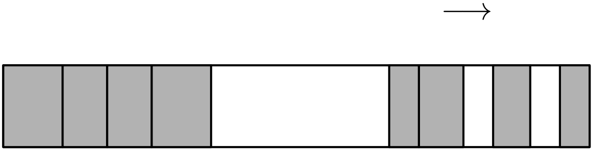
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We slide every live ordinary object **from right to left**.

Proposed algorithm: compaction

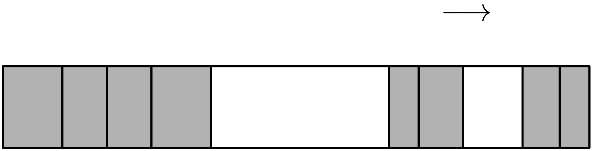
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We slide every live meta-object **from left to right**.

Proposed algorithm: compaction

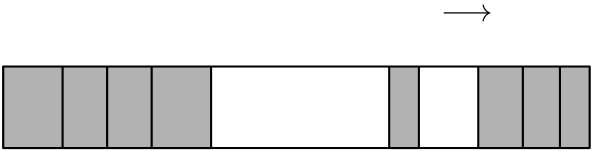
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We slide every live meta-object **from left to right**.

Proposed algorithm: compaction

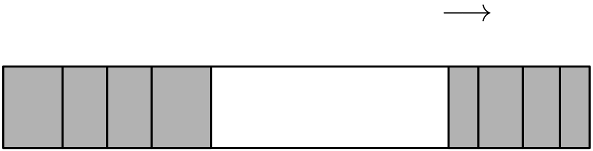
	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$



We slide every live meta-object **from left to right**.

Proposed algorithm: compaction

	Ordinary objects	Meta-objects
Allocation	$\longrightarrow$	$\longleftarrow$
Scanning	$\longrightarrow$	$\longleftarrow$
Sliding	$\longleftarrow$	$\longrightarrow$

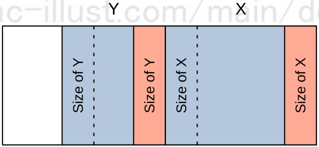


GC is now complete.

Boundary tag

Fusuma scans the meta-object area from right to left.
⇒ Fusuma places size information at both ends of every meta-object.

This size information at the bottom of a meta-object is called the **boundary tag**.

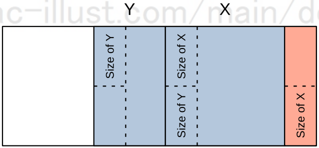


A naive implementation is to add an extra word next to a meta-object.
⇒ This implementation imposes space overhead.

Boundary tag

Fusuma scans the meta-object area from right to left.
⇒ Fusuma places size information at both ends of every meta-object.

This size information at the bottom of a meta-object is called the **boundary tag**.



An embedding implementation halves the bit length of the size information in the meta-object header.
⇒ This implementation merges the boundary tag of a meta-object with the header of the immediately following one.

Evaluation

We implemented Fusuma in eJSVM and evaluated it.

Environment

	Intel x64 (X64)	Raspberry Pi (RP)
CPU	Core i7-10700	Cortex-A53 (ARMv8)
Frequency	2.90 GHz	1.40 GHz
OS	Debian 10.7	Raspbian 9.13
eJSVM	for 64bit	for 32bit

We compared three eJSVMs that used different GC algorithms.

MS : the mark-sweep algorithm

TC : Fusuma using the naive boundary tag implementation

TCE : Fusuma using the boundary tag embedding implementation

Benchmark programs

Benchmark programs:

- From AreWeFastYet benchmark: 7 programs
- From Sunspider benchmark: 8 programs
- IoT application: 1 program
  - repeatedly converts a sequence of bits from a temperature and humidity sensor into numerical values.
- Synthetic program: 1 program
  - continuously adds new properties.

Benchmark programs can be classified into two groups.

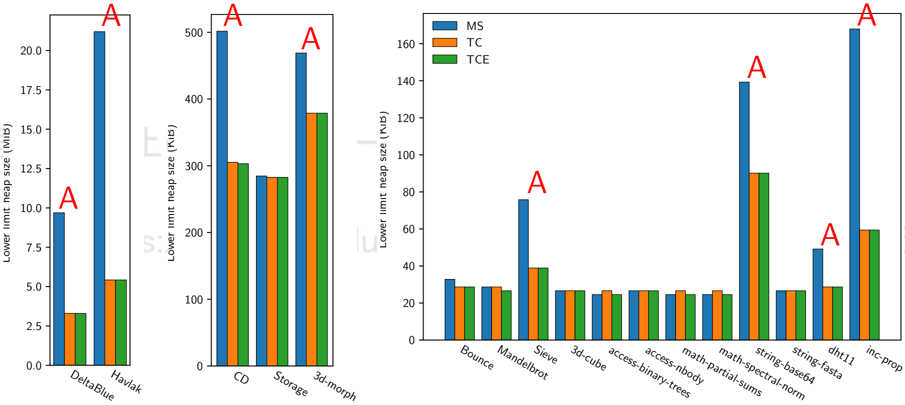
- Group A: Fragmentation occurred in MS: 8 programs.
- Group B: Serious fragmentation did not occur in MS: 9 programs.

Evaluation items

- Space efficiency:
  - the lower limit heap size required to run each program

- Time efficiency:
  - execution times and GC times of each program against the heap sizes

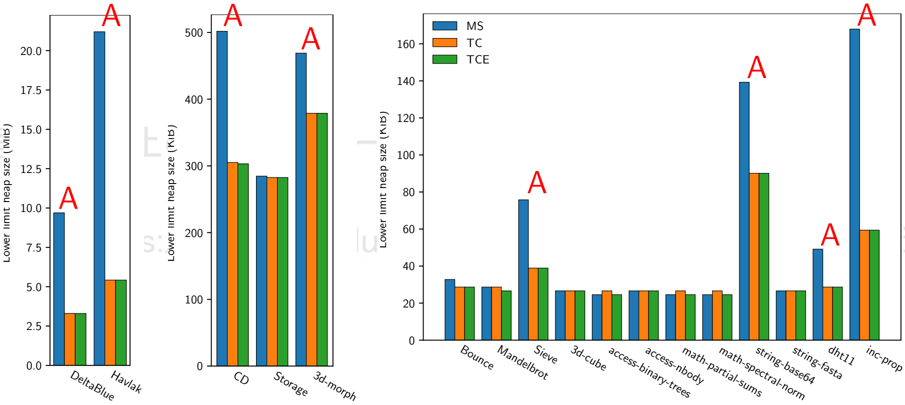
Space efficiency: lower limit heap size (RP)



Group A:

- The lower limits for TC and TCE were substantially smaller than that for MS.
  $\Rightarrow$ The fragmentation occurred in MS was eliminated by the compaction of TC and TCE.

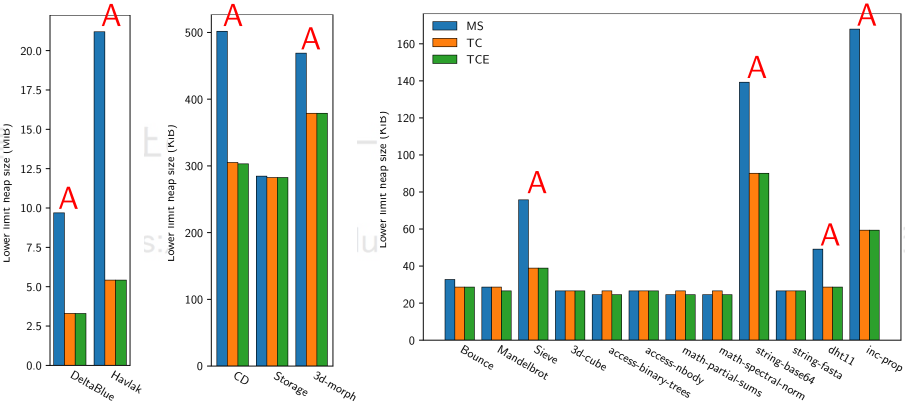
Space efficiency: lower limit heap size (RP)



Group A:

- For an IoT-oriented program, TCE reduced the lower limit by 20 KiB (40%) compared with MS.

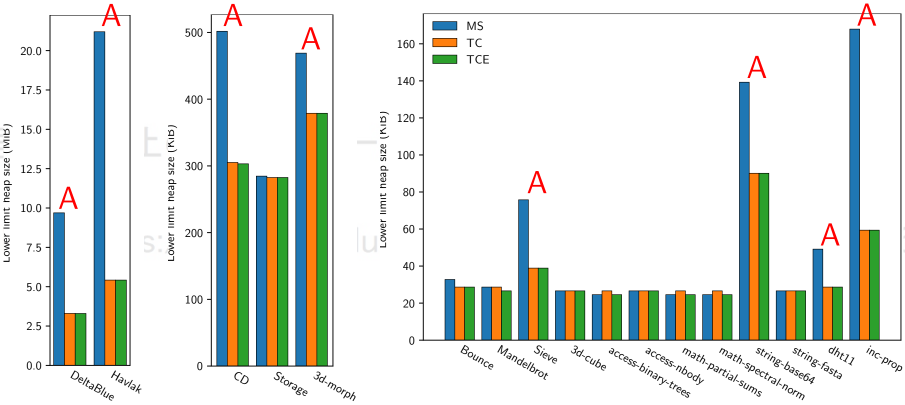
Space efficiency: lower limit heap size (RP)



Group B:

- The lower limits for MS and TCE were similar.

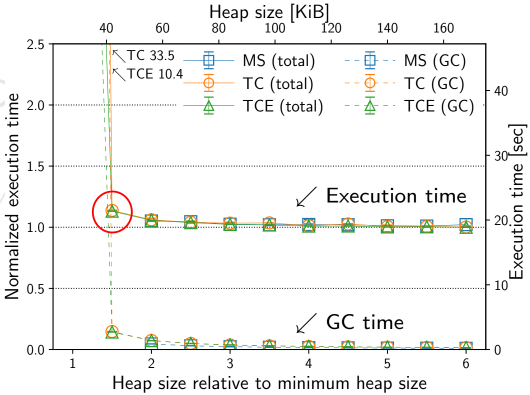
Space efficiency: lower limit heap size (RP)



Most programs in A and B:

- The lower limit for TC was larger than that for TCE.
  $\Rightarrow$ The spatial overhead of the boundary tags was successfully reduced in TCE.

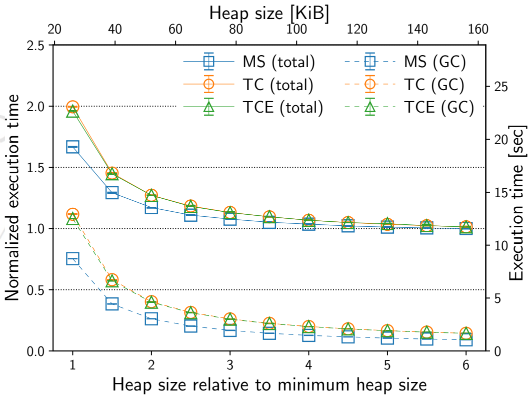
Time efficiency: dht11 (RP, group A)



※ lower is better

- MS failed to run at 1.5x the minimum heap size, while TC and TCE ran in a reasonable time.

Time efficiency: access-nbody (RP, group B)

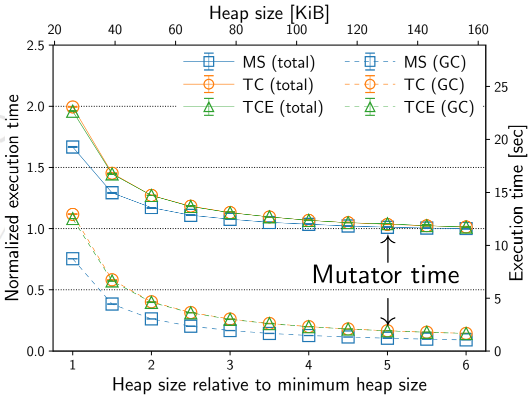


※ lower is better

Small heap:
MS ran faster than TC and TCE.

- TC and TCE took longer GC time due to compaction.

Time efficiency: access-nbody (RP, group B)



※ lower is better

Larger heap (6x the minimum heap size):
All three showed almost the same performance.

- GC time: MS $<$ TC $\approx$ TCE
- Mutator time: MS $>$ TC $\approx$ TCE due to improvement of locality?

Overhead of GC (1)

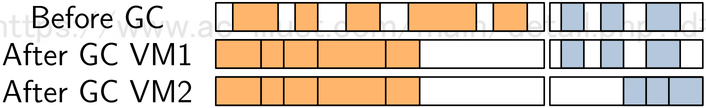
In many programs, TCE took more GC time than MS.

⇒ This overhead was caused by the use of threaded compaction itself, **not by the use of Fusuma** (double-ended method).

To confirm this, we conducted experiments on the next slide.

Overhead of GC (2)

- We prepared two separate heaps, one for ordinary objects and the other for meta-objects.
- We compared two variations.
  - VM1 executed GC only for ordinary object area.
  - VM2 executed GC for both heaps.



Ratio of GC times (VM1/VM2)

	X64	RP
Minimum	0.88	0.87
Average	0.96	0.95

Compaction of ordinary object area was the dominant factor of the GC time of Fusuma.

Related work: Other approaches to this problem

Manage meta-objects in a separate heap :

- MMTk [Blackburn, et al.'04]
  → It may space level fragmentation.

Move to unused area, avoiding overwriting 'from-object' :

- Copying GC [Cheney'70]
  → It needs 'copy reserve'.

Fusuma is more space efficient.

Related work: Other sliding compactions

Other compaction algorithms than Jonkers's also cause the problems focused on in this research.

Major sliding compaction algorithms :

- Lisp2 [Knuth'97]
  - Requires additional space for storing forwarding pointer in every object.
- Break-Table [Haddon, et al.'67]
  - Needs to sort object destination table.

Jonkers's algorithm requires no additional space, and needs only scanning heap with threading.

Conclusion

- We have proposed Fusuma, a double-ended threaded compaction.
    - This allows ordinary objects and meta-objects to be allocated in the same heap.
- By using the boundary tag embedding, Fusuma can be implemented without any extra space for each meta-object.
- We implemented Fusuma in eJSVM and confirmed its effectiveness.

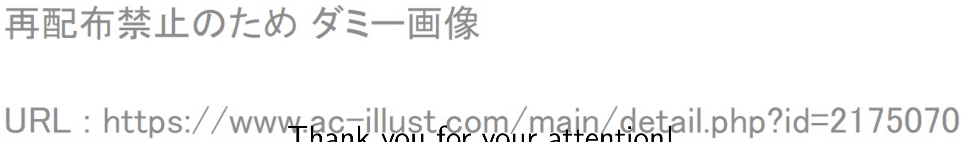
Thank you!

再配布禁止のため ダミー画像

URL : https://www.ac-illust.com/main/detail.php?id=2175062

This is all of my presentation.

Thank you!



Thank you for your attention!