Fusuma: Double-Ended Threaded Compaction (preprint)

Hiro Onozawa
Graduate School of Informatics and Engineering
The University of Electro-Communications
Tokyo, Japan
19honozawa@ipl.cs.uec.ac.jp

Tomoharu Ugawa
Graduate School of Information Science and Technology
The University of Tokyo
Tokyo, Japan
tugawa@acm.org

Hideya Iwasaki
Graduate School of Informatics and Engineering
The University of Electro-Communications
Tokyo, Japan
iwasaki@cs.uec.ac.jp

Abstract

Jonkers's threaded compaction is attractive in the context of memory-constrained embedded systems because of its space efficiency. However, it cannot be applied to a heap where ordinary objects and meta-objects are intermingled for the following reason. It requires the object layout information, which is often stored in meta-objects, to update pointer fields inside objects correctly. Because Jonkers's threaded compaction reverses pointer directions during garbage collection (GC), it cannot follow the pointers to obtain the object layout. This paper proposes Fusuma, a double-ended threaded compaction that allows ordinary objects and meta-objects to be allocated in the same heap. Its key idea is to segregate ordinary objects at one end of the monolithic heap and meta-objects at the other to make it possible to separate the phases of threading pointers in ordinary objects and meta-objects. Much like Jonkers's threaded compaction, Fusuma does not require any additional space for each object. We implemented it in eJSVM, a JavaScript virtual machine for embedded systems, and compared its performance with eJSVM using mark-sweep GC. As a result, compaction enabled an IoT-oriented benchmark program to run in a 28-KiB heap, which is 20 KiB smaller than mark-sweep GC. We also confirmed that the GC overhead of Fusuma was less than 2.50× that of mark-sweep GC.

Keywords: garbage collection, JavaScript, compaction, meta-object, embedded systems

1 Introduction

eJS [20] is a JavaScript processing system for embedded systems such as IoT devices, where approximately 100 KB of memory is available for the heap. It generates a customized JavaScript virtual machine (VM), eJSVM, for a specific application based on a VM configuration given by the application developer.

The standard eJSVM configuration uses mark-sweep garbage collection (GC). Because mark-sweep GC does not move objects, the VM may suffer from fragmentation; even if the total free space in the heap is sufficient, the program may fail to allocate a contiguous area for an object. One way to solve this fragmentation problem is to use compaction. Jonkers's threaded compaction [12] is a memory-efficient compaction algorithm. Unlike other sliding compaction methods, it does not use extra words in the object header or in any side tables. For example, Lisp2 [14] compaction uses one extra word for a forwarding pointer, and the Compressor [13] uses a side table. Because space overhead of a few kilobytes is serious in a 100-KB heap, this attribute is attractive.

The current eJSVM records the layout information of JavaScript first class objects, or *ordinary objects*, in *meta-objects* called *hidden classes* [4] to execute JavaScript programs efficiently. Because a hidden class is made up of multiple meta-objects connected by pointers, the pointers inside meta-objects must be followed to obtain the object layout. Such meta-objects are allocated in the same heap as ordinary objects and are subject to GC.

Jonkers's threaded compaction scans the heap after marking live objects to *thread* all pointers in the object. The layout of each object must be known during the threading process. However, because the threading operation reverses the pointer direction, threaded pointers cannot be followed until they are unthreaded. Because of this, if ordinary objects and meta-objects are allocated in the same heap, the object layout cannot be obtained.

One possible way to cope with this problem is to use a dedicated space for meta-objects that is separate from the space for ordinary objects. However, separate spaces might induce space-level fragmentation, where one space becomes full while the other has an excess of free space. This might be fatal in memory-constrained environments.

ISMM '21, June 22, 2021, Virtual, Canada

This paper proposes *double-ended threaded compaction* named Fusuma[1] to resolve this problem. Fusuma distinguishes between ordinary objects and meta-objects when allocating them: ordinary objects are allocated from one end of the monolithic heap and the meta-objects that make up the hidden classes from the other end. During compaction, pointers in ordinary objects are threaded first, and then the meta-objects, which have a statically determined layout, are threaded. By threading in this order, GC can follow pointers in meta-objects to obtain the layout of ordinary objects because the meta-objects have not yet been threaded during ordinary object processing. Finally, live ordinary objects and meta-objects are slid toward the left and the right respectively. As in Jonkers's algorithm, Fusuma does not require any additional space.

We implemented Fusuma in eJSVM and evaluated its performance using several benchmark programs. Compacting the heap enabled an IoT-oriented benchmark program to run in a 28-KiB heap, which is 20 KiB smaller than for mark-sweep GC. We also confirmed that the GC overhead of Fusuma was less than 2.50× that of mark-sweep GC.

The problem addressed in this paper is not specific to eJSVM. For example, in the V8 JavaScript Engine[2], the GC needs to follow pointers in hidden classes to determine whether each property is a pointer or an unboxed value [7]. Implementing Jonkers's compaction in such a system involves the same problem as eJSVM, which can be solved by Fusuma. Furthermore, other in-place sliding compactions [10, 14] share essentially the same problem addressed in this paper. The central idea of Fusuma, sliding objects in two directions, can be applied to them to solve the problem, as discussed in Section 7.

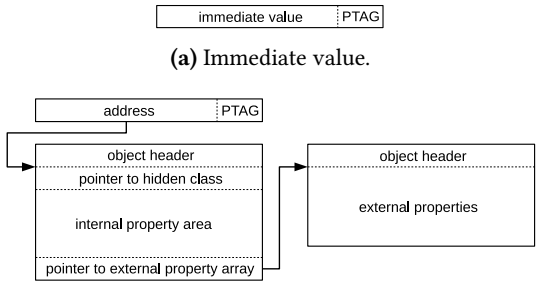
This paper first describes eJS in Section 2, followed by Jonkers's compaction algorithm and its problems in Section 3. Next, Section 4 proposes Fusuma, a double-ended threaded compaction algorithm, and Section 5 describes its implementation issues in eJSVM. Section 6 evaluates the performance of Fusuma implemented in Section 5 by means of experiments. Section 7 introduces related work, and finally, Section 8 concludes the paper.

2 eJS

2.1 Overview

eJS (embedded JavaScript) [20] is a JavaScript processing system for embedded systems such as IoT devices, where approximately 100 KB of memory is available for the heap. It reduces the workload of program developers, improves application development efficiency, and facilitates prototyping by enabling development on embedded systems in JavaScript.

[1]Fusuma is a bi-part sliding partition in a traditional Japanese room.
[2]https://v8.dev/



(a) Immediate value.

(b) Pointer.

Figure 1. `JSValue` in eJSVM.

The eJS framework can automatically generate eJSVM, which is a JavaScript VM customized for a specific application program to run on a specific embedded system. For example, the developer can specify datatypes to be distinguished with pointer tagging. eJS generates efficient type dispatching code in eJSVM based on the developer's specification. The eJS framework supports both 32-bit and 64-bit processors. Currently, eJSVM runs with an interpreter without just-in-time (JIT) compilation.

2.2 Object Layout

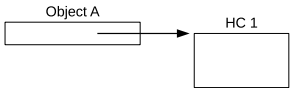
In eJSVM, every first-class value in JavaScript is represented by a single-word `JSValue` type. Its structure is presented in Figure 1. A value of `JSValue` has type information called a PTAG (pointer tag) and either an immediate value or an object address in a single word. When a value of the `JSValue` type is an integer, a boolean, or a special value such as `undefined`, an immediate value is stored. When it represents an ordinary object such as an `Object` or an `Array`, the address of the object is used. Because every object is word-aligned in the heap area, the lower two or three bits (depending on word size) of an object address are always zero. These bits are used for PTAG. The developer can specify a few datatypes that have separate PTAG values to examine the types quickly. Other datatypes share the same PTAG values and can be distinguished by the type information called the HTAG (header tag) in the object header described below.

An ordinary object consists of four elements: an *object header*, a pointer to a hidden class, an internal property area, and a pointer to an external property array, if any. The object header is a single word containing the object's size, type (HTAG), and the mark bit used in GC. The hidden class records the object layout. An object reference points to the next word to the object header, as presented in Figure 1 (b).

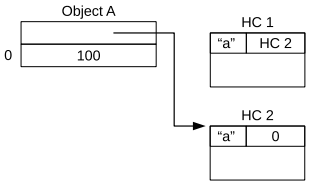
Invisible properties from the JavaScript program that are inherently possessed by an ordinary object when it is created, such as the pointer to the code of a built-in function, are called *special properties*. Associated values with special properties are stored as non-`JSValues` at the beginning of the internal property area, with some exceptions, which GC takes care of in an ad-hoc way.

```
var A = {};    // Step 1
A.a = 100      // Step 2
A.b = "Hello" // Step 3
var B = {};    // Step 4
B.a = 200      // Step 5
```

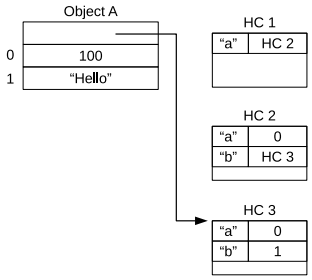
(a) JavaScript code.



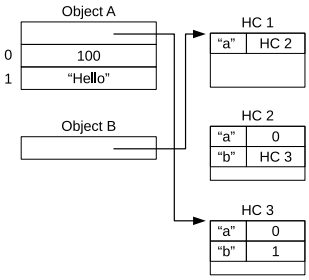
(b) Step 1.



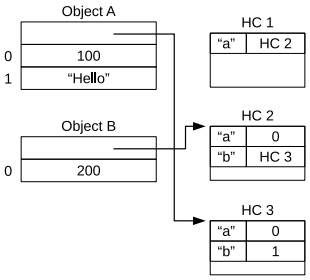
(c) Step 2.



(d) Step 3.



(e) Step 4.



(f) Step 5.

Figure 2. Example of growing hidden classes.

As in other JavaScript VMs [6], properties that are likely to be added to an ordinary object in the future are reserved in advance as internal properties when the object is allocated. They are determined based on the execution history of the program up to the point where the object is allocated. Other properties that are attached dynamically during program execution are stored in the external property array.

2.3 Hidden Classes

JavaScript can dynamically add and remove the properties of objects. A naive implementation for such dynamic addition and deletion of properties uses an array or hash table of key-value pairs, where the key is a property name, and the value is its associated value. However, this naive implementation is not efficient.

A common technique to deal efficiently with JavaScript's dynamic behavior is to use *hidden classes* [4]. When using hidden classes, each object manages only its property values in an array, which is called the *property array*. In addition to this, each object has a pointer to its hidden class. A hidden class essentially manages mappings from property names to *indices* in the property array. Hidden classes are immutable; they can be shared among multiple objects with the same set of property names that have been attached in the same order. When a new property is added to an object, a new hidden class containing both the existing properties and the new property is created. This new hidden class is linked by a transition edge from the previously used hidden class. The transition edge is followed to find the new hidden class when the property of the same name is added to another object that shares the previously used hidden class. Figure 2 shows an example of hidden classes that grow as properties are added to objects.

Hidden classes can improve space efficiency because the information on where object properties are stored is shared among multiple objects. In addition, access to properties can be accelerated by using inline caching [9, 11], which premises hidden classes.

2.4 Implementation of Hidden Classes in eJSVM

A hidden class in eJSVM consists of two kinds of meta-objects: one is the *Layout*, and the other includes the *PropertyMaps* and their related objects. Each meta-object has a similar header to that of an ordinary object.

Figure 3 shows the structure of these meta-objects. A Layout represents the memory layout of ordinary objects that hold it. More specifically, it represents the size of the internal property area and that of the external property array. A PropertyMap is essentially a table representing the mappings from property names to the locations of their associated values or transition edges. The second slot of the PropertyMap in Figure 3 holds the pointer to the table, which is omitted in the figure. A PropertyMap also contains the total number of properties and the number of special properties.

Each ordinary object could have a Layout and a PropertyMap. Instead, eJSVM saves space by putting the pointer to the PropertyMap in the Layout. In Figure 3, `Object1` and `Object2` have different numbers of internal properties. Therefore, their Layouts are different, but they share the same PropertyMap.

The GC refers to these meta-objects to obtain the sizes of the internal property area and the external property array stored in the Layout and to obtain the number of special properties, which is stored in the PropertyMap. Specifically, GC needs to follow the pointer to the PropertyMap stored in the Layout to obtain the number of special properties.

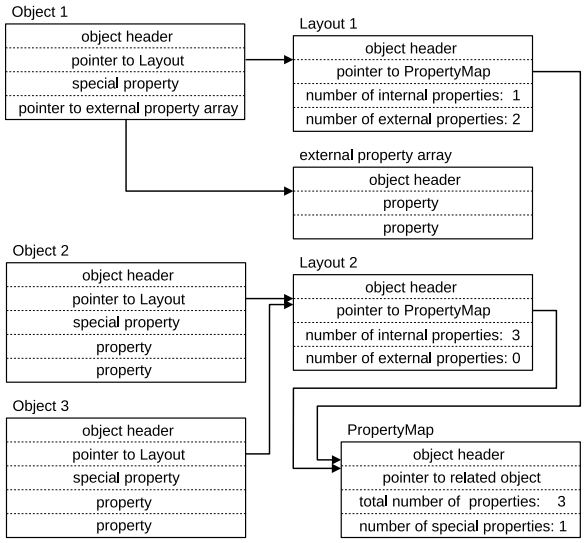


Figure 3. Structure of a hidden class in eJSVM.

There are cases where meta-objects become garbage and are reclaimed by GC in eJSVM. For example, a prototype object is unlikely to share its PropertyMap with other objects because each prototype object is usually created only once. Therefore, PropertyMaps created during initialization of prototype objects are likely to be tentative. The eJSVM unlinks such tentative PropertyMaps from the transition edge. The details of this are beyond the scope of this paper.

3 Jonkers's Threaded Compaction

3.1 Overview

The threaded compaction algorithm by Jonkers [12] performs space-efficient sliding compaction. This algorithm transforms multiple pointers to the same object into a linear list of locations for the pointers by reversing the pointer direction through an operation called *threading*. Thanks to the threading operation, Jonkers's algorithm does not require any extra space in an object, as opposed to the Lisp2 algorithm [14].

Figure 4 (a)–(b) presents the relation between pointers and objects before and after threading. After threading, the header area of the object becomes the head of the list of locations that pointed to the object. This list of threaded pointers is called the *threaded list* from the object. The end of the threaded list contains the original value in the object header ("X" in Figure 4). Jonkers's algorithm must recognize that it does not hold a threaded pointer to detect the end of a threaded list.

After determining an object's destination, to which the object is to be moved, the algorithm *unthreads* the threaded list to update every location that pointed at the object before threading with the destination address, as presented in Figure 4 (c). The pointer locations to be updated can be found by following the threaded list from the object header.

3.2 Algorithm

Jonkers's compaction algorithm consists of three phases: the *mark* phase, the *update-forward-reference* phase, and the *update-backward-reference* phase. Figure 5 presents an example of a heap when this algorithm is applied. The two pointers on the left of each sub-figure represent the roots, and the wide area on their right represents the heap. The value in the header of an object, e.g., a, is used to name the object itself, for example, "object a."

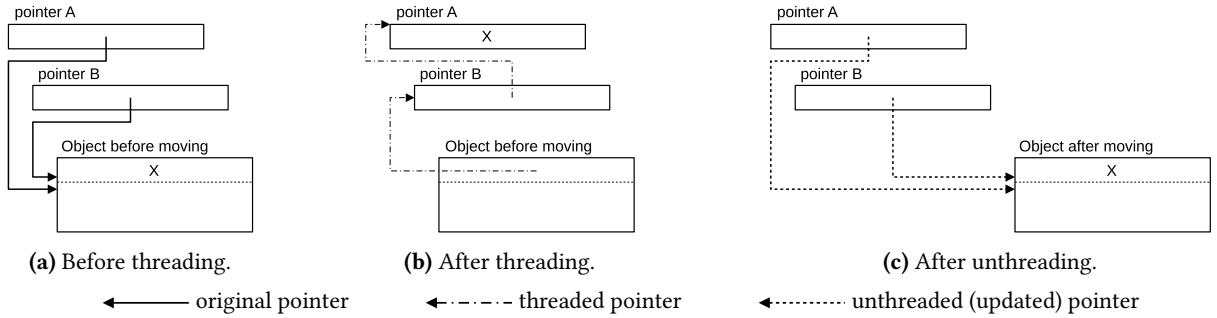
First, the mark phase marks all live objects, which are reachable from the *roots*. The roots are pointers to objects in the heap from outside. The heap after the mark phase is presented in Figure 5 (a). In this figure, the gray color of objects indicates that they are alive.

Next, the update-forward-reference phase updates every pointer that points to a live object forward in the heap with the destination address, to which the object is to be moved (Figure 5 (b)–(f)). First, this phase threads all pointers in the roots. It then scans the heap from the left to find live objects and performs the following operations on every live object. First, it unthreads the threaded list from the object by updating every location in the list with its destination address. Because the heap is scanned from the left, the destination address is determined by summing up the sizes of all live objects found so far. Next, the phase threads all pointers in the object. The heap area after the update-forward-reference phase is presented in Figure 5 (f), where only backward references remain threaded.

Finally, the update-backward-reference phase moves all live objects while unthreading backward pointers (Figure 5 (g)–(j)). As in the update-forward-reference phase, the update-backward-reference phase scans the heap area from the left and performs the following operations on every live object. First, it unthreads the threaded list from the object by updating every location in the list with its destination address. Then it moves the object to the destination address. The heap area after this phase is presented in Figure 5 (j), where all live objects have been slid toward the left of the heap and all pointers have been correctly updated.

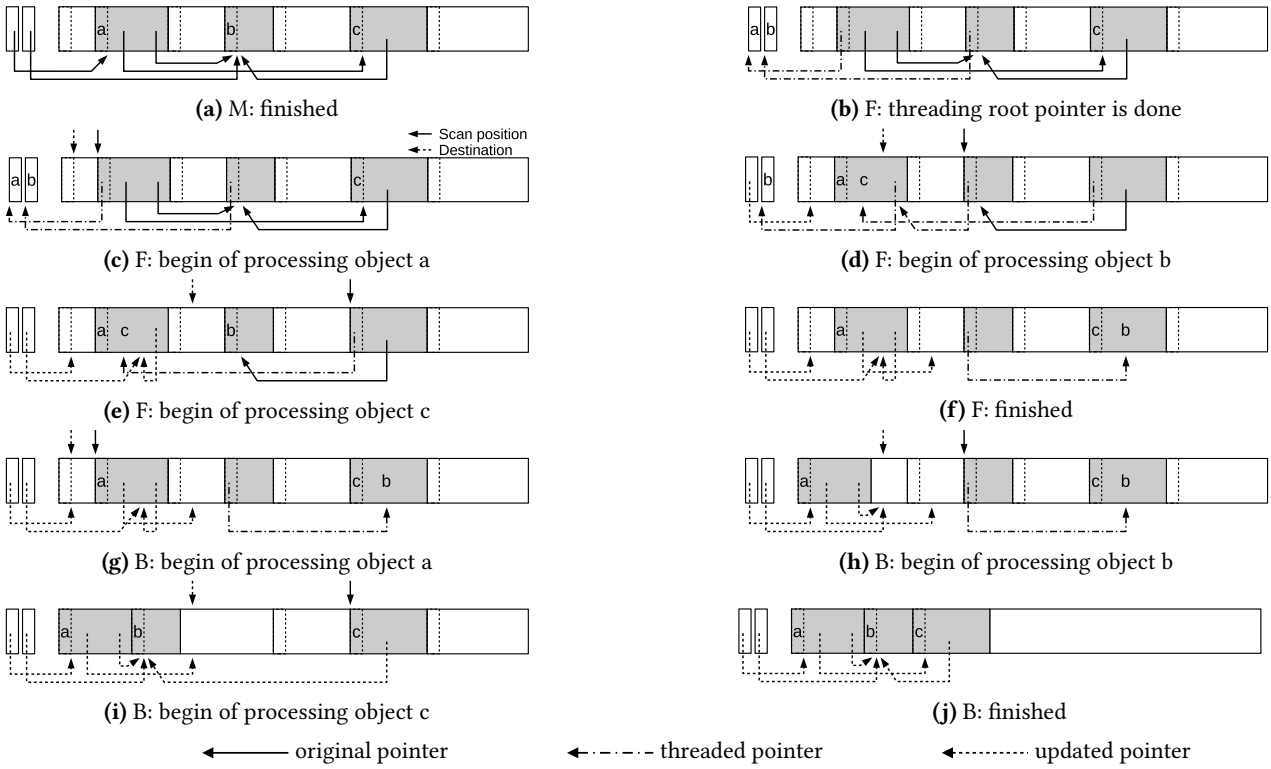
3.3 Problem

During the mark and update-forward-reference phases, the layout information of every object is indispensable to know the locations of pointer fields in the object. In eJSVM, the numbers of internal and external properties in the Layout and the number of special properties in the PropertyMap are necessary. If eJSVM implements unboxing [7] in the future, information about whether each field holds a `JSValue` or an unboxed value will also be stored in PropertyMaps. Unfortunately, when both meta-objects that constitute hidden



(a) Before threading. **(b)** After threading. **(c)** After unthreading.

Figure 4. Threading and unthreading.



(a) M: finished **(b)** F: threading root pointer is done

(c) F: begin of processing object a **(d)** F: begin of processing object b

(e) F: begin of processing object c **(f)** F: finished

(g) B: begin of processing object a **(h)** B: begin of processing object b

(i) B: begin of processing object c **(j)** B: finished

M: mark phase, F: update-forward-reference phase, B: update-backward-reference phase

Figure 5. Example of heap with Jonkers's threaded compaction applied.

classes and ordinary objects are intermingled in the same heap, the following problem might occur.

Suppose that an ordinary object, its Layout, and its PropertyMap reside in the heap in the order of the Layout, the object, and the PropertyMap, as presented in Figure 6 (a). In the update-forward-reference phase, the pointer in the Layout to the PropertyMap is threaded first (Figure 6 (b)). Then, when processing the object, it is possible to reach the Layout from the object, but the number of special properties in the PropertyMap is inaccessible because the pointer from the Layout to the PropertyMap has already been threaded. As a result, it is impossible to know the layout of the object, which prevents Jonkers's algorithm from being applied.

4 Double-Ended Threaded Compaction

This paper proposes Fusuma, a *double-ended threaded compaction* algorithm that solves the problem described in Section 3.3.

Assume that the system has ordinary objects and meta-objects and that they reside in the same monolithic heap. GC may need to *follow* pointers in meta-objects to obtain the layout information of ordinary objects. It does not matter whether meta-objects have pointers to ordinary objects if GC does not follow them. For example, hidden classes have pointers to string objects, which are ordinary objects, in eJSVM, but GC does not follow the pointers.

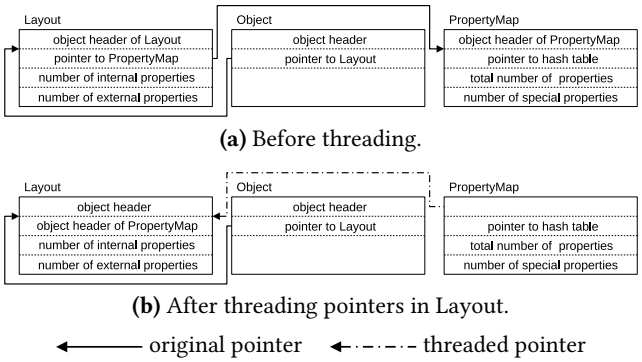


Figure 6. Problem in Jonkers's algorithm.

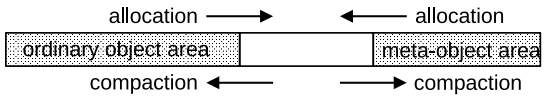


Figure 7. Heap usage of Fusuma.

4.1 Basic Idea

Fusuma has the same phases as Jonkers's compaction.

The basic idea of the proposed algorithm is that, in the update-forward-reference phase, it processes all ordinary objects first, during which it refers to meta-objects for layout information to determine the locations of pointer fields inside ordinary objects. Afterwards, it processes meta-objects. Note that the layout of every meta-object is statically determined.

For this purpose, ordinary objects are allocated from the left of the heap in the forward direction and meta-objects from the right in the backward direction (Figure 7). The heap area where ordinary objects are allocated is called the *ordinary object area*, and that where meta-objects are allocated is called the *meta-object area*.

During the compaction, ordinary objects and meta-objects are slid toward the left end and the right end of the heap respectively. Therefore, in the update-forward-reference phase and update-backward-reference phase, ordinary objects are scanned from the left, whereas meta-objects are scanned from the right.

4.2 Algorithm

Figure 8 presents the pseudo code of Fusuma. In this code, `metaObjectTop(to,size)` returns the top address of the meta-object of which the bottom is pointed to by `to`, and `pointerToObject(p)` returns the pointer to the ordinary object / meta-object with a top address of `p`. For the case of eJSVM, `p` points to the object header, and this function returns the address of the next word to the header. Function `getMetaObjectSize(b)` determines the size of a meta-object from its bottom address `b`. As for the other helper functions, their names are self-explanatory. Figure 9 provides an example of a heap when Fusuma is applied.

Much like Jonkers's compaction, Fusuma consists of three phases: the mark phase, the update-forward-reference phase, and the update-backward-reference phase. The mark phase is the same as that of Jonkers's compaction.

The update-forward-reference phase can be divided into three steps (function `updateForwardReferences()`). The first step (**`foreach`** loop) threads all pointers in the roots (Figure 9 (a)–(b)). The second step (the first **`while`** loop) scans the ordinary object area from the left in the forward direction, as in the update-forward-reference phase of Jonkers's algorithm (Figure 9 (b)–(c)). Finally, the third step (the second **`while`** loop) scans the meta-object area from the right in the reverse direction (Figure 9 (c)–(e)). The processing for each live meta-object found during the third step is the same as for each live ordinary object in the second step; first, the algorithm unthreads the threaded list by updating every location in the list with the new destination address, and then it threads every pointer in the meta-object.

The update-backward-reference phase (function `updateBackwardReferences()`) scans (in the first **`while`** loop) the object area in the forward direction (Figure 9 (e)–(f)), and then scans (in the second **`while`** loop) the meta-object area in the reverse direction (Figure 9 (f)–(h)). The operation for each ordinary object / meta-object is the same as the operation performed in the update-backward-reference phase in Jonkers's algorithm; it unthreads the threaded list and then moves the object to its destination address. Please note that an ordinary object is moved toward the left of the heap, whereas a meta-object is moved toward the right.

By processing all ordinary objects before threading the pointers inside meta-objects in the update-forward-reference phase, the problem described in Section 3.3 is solved.

5 Implementation

5.1 Boundary Tags

Fusuma scans the meta-object area from right to left. To make this possible, Fusuma places size information not only in the header of every meta-object, but also in the next word of the meta-object. This size information at the bottom of a meta-object is called the *boundary tag*. Although no left-to-right iteration is performed on the meta-objects, the object header, including size and type, is still needed at the left of each meta-object to make it possible to obtain its type in the same manner as for ordinary objects.

A naive implementation of the boundary tag is to add an extra word next to a meta-object, as presented in Figure 10 (a). This implementation imposes space overhead, which might be critical for embedded systems with limited memory for the heap. To reduce this overhead, this paper proposes to use the *boundary tag embedding* technique [14].

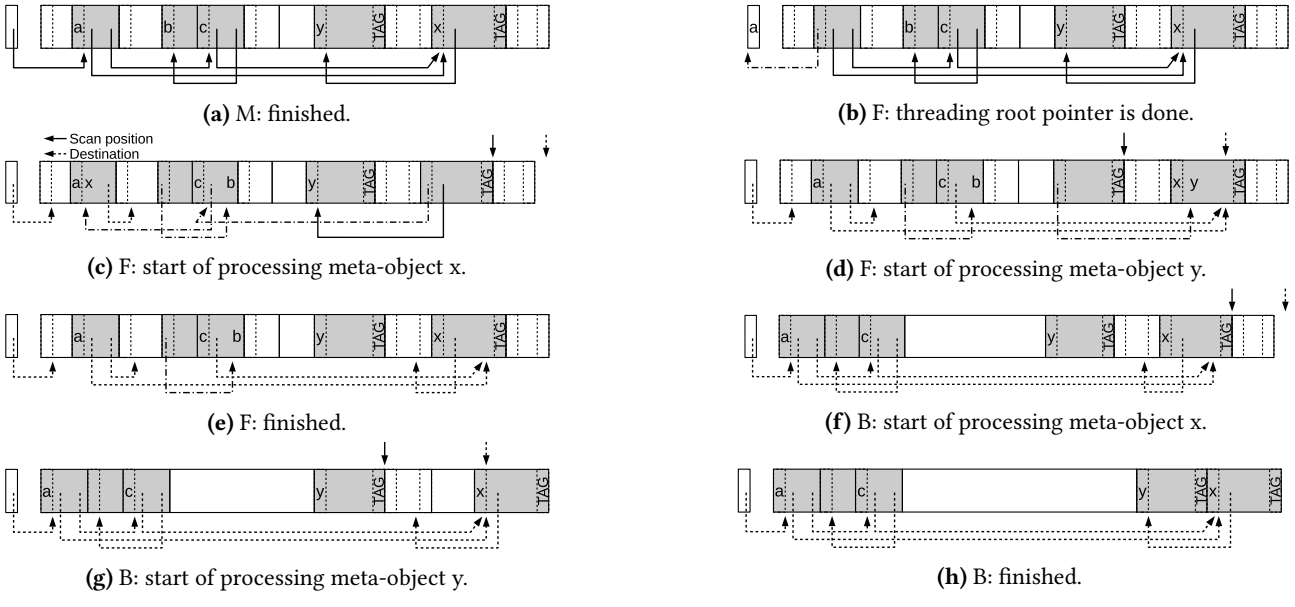
The boundary tag embedding technique halves the bit length of the size information in the meta-object header and merges the boundary tag of a meta-object with the header

```
gc() {
  mark();
  updateForwardReferences();
  updateBackwardReferences();
}

thread(ref) {
  ptr = *ref;
  *ref = getHeader(ptr);
  setHeader(ptr, ref);
}

unthread(ptr, dst) {
  tmp = getHeader(ptr);
  while (isThreadedPointer(tmp)) {
    next = *tmp;
    *tmp = dst;
    tmp = next;
  }
  setHeader(ptr, tmp);
}

threadAllPointers(p) {
  foreach (ptr in getPointers(p))
    thread(&ptr)
}
```

```
updateForwardReferences() {
  foreach (ptr in roots)
    thread(&ptr)
  scan = to = ordinaryObjectAreaStart;
  while (scan < ordinaryObjectAreaEnd) {
    p = pointerToObject(scan);
    size = getObjectSize(p)
    scan += size;
    if (isMarked(p)) {
      dest = pointerToObject(to);
      unthread(p, dest);
      threadAllPointers(p);
      to += size;
    }
  }
  scan = to = metaObjectAreaEnd;
  while (metaObjectAreaStart < scan) {
    size = getMetaObjectSize(scan);
    scan = metaObjectTop(scan, size);
    p = pointerToObject(scan);
    if (isMarked(p)) {
      to = metaObjectTop(to, size);
      dest = pointerToObject(to);
      unthread(p, dest);
      threadAllPointers(p);
    }
  }
}
```

```
updateBackwardReferences() {
  scan = to = ordinaryObjectAreaStart;
  while (scan < ordinaryObjectAreaEnd) {
    p = pointerToObject(scan);
    size = getObjectSize(p)
    scan += size;
    if (isMarked(p)) {
      dest = pointerToObject(to);
      unthread(p, dest);
      move(p, dest);
      to += size;
    }
  }
  scan = to = metaObjectAreaEnd;
  while (metaObjectAreaStart < scan) {
    size = getMetaObjectSize(scan);
    scan = metaObjectTop(scan, size);
    p = pointerToObject(scan);
    if (isMarked(p)) {
      to = metaObjectTop(to, size);
      dest = pointerToObject(to);
      unthread(p, dest);
      move(p, dest);
    }
  }
}
```

Figure 8. Pseudo code of Fusuma.



(a) M: finished.

(b) F: threading root pointer is done.

(c) F: start of processing meta-object x.

(d) F: start of processing meta-object y.

(e) F: finished.

(f) B: start of processing meta-object x.

(g) B: start of processing meta-object y.

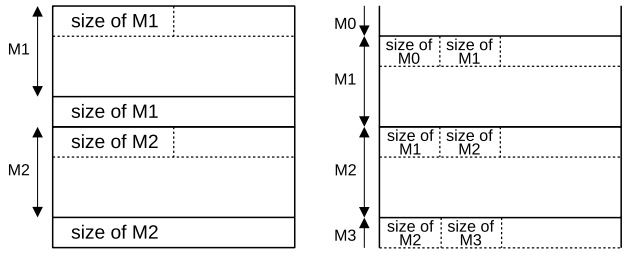
(h) B: finished.

M: mark phase, F: update-forward-reference phase, B: update-backward-reference phase

Figure 9. Example of heap with Fusuma applied.

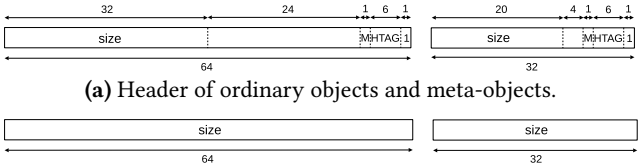
of the immediately following meta-object, as presented in Figure 10 (b). Through this technique, Fusuma can be implemented without any extra space overhead except for a single word containing the boundary tag for the rightmost meta-object.

One drawback of the boundary tag embedding is that it sacrifices the size field in the meta-object header. However, in practical use in eJSVM, this is unlikely to introduce a problem because it does not affect the size of ordinary objects, and meta-objects are usually small enough. As will be presented



(a) Naive implementation. **(b)** Embedding implementation.
M0–M3 represent meta-objects.

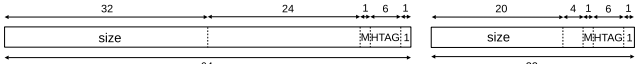
Figure 10. Two implementations of boundary tag.



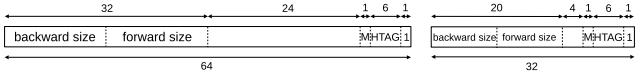
(a) Header of ordinary objects and meta-objects.

(b) Boundary tag of meta-objects.

Figure 11. Header and boundary tag in the naive implementation (M: mark bit, left: 64-bit, right: 32-bit).



(a) Header of ordinary objects.



(b) Header of meta-objects.

Figure 12. Header in boundary tag embedding implementation (M: mark bit, left: 64-bit, right: 32-bit).

in Section 5.2, even for the 32-bit implementation, the size field has 20 bits. After the size field is halved, it still has 10 bits, allowing up to 1,023 words. In eJSVM, meta-objects do not exceed this limit unless an object has more than 510 properties or a PropertyMap has more than 510 transition links. In fact, the maximum size of meta-objects was only 163 words in the benchmark programs in Section 6. Nevertheless, large meta-objects could be supported, although this has not yet been implemented, by putting some special marker value in the size field and recording the actual (large) size in an adjacent extra word.

5.2 Object Header Types

Figure 11 presents the object header of both ordinary objects and meta-objects and the boundary tag of the meta-objects used in the naive implementation. Figure 12 presents the object header of the ordinary objects and that of the meta-objects used in the boundary tag embedding implementation.

For boundary tag embedding, the size field in the meta-object header is equally divided into two: the *forward size* field that holds the meta-object size of the header, and the *backward size* field that holds the meta-object size just before the header. Therefore, the bit length of the size field in the ordinary object header is different from that of the forward size field in the meta-object header. Despite this, when taking the size of the current object of interest from its header, there is no need to determine at runtime whether this object is an ordinary object or a meta-object to use the proper bitmask. Instead, it is always possible to use a deterministic bitmask depending on the scanning area because the current object is definitively an ordinary object / a meta-object when scanning the ordinary object area / meta-object area respectively.

5.3 Terminal Bit

For both implementations, the LSB of every header is the *terminal bit*, which is always set. As explained in Section 3, distinguishing between a threaded pointer and a header is necessary. The terminal bit is used for this purpose.

Pointers to ordinary objects / meta-objects are expected to be placed at even addresses in memory. They may reside on the GC target heap where ordinary objects and meta-objects are allocated, or outside the heap, such as local variables in a C function. In the former case, pointers are always placed at even addresses because objects are aligned at 64-bit / 32-bit boundaries. In the latter case, although pointers are not guaranteed to be placed at even addresses, the C compiler generally avoids placing them at odd addresses. Therefore, the LSB of a threaded pointer is cleared in our implementation of Fusuma. The `unthread` in Figure 8 investigates the LSB of a target value by using `isThreadedPointer`. If the LSB is cleared, the value is judged to be a threaded pointer; otherwise, the value is a header.

5.4 Restoring the Pointer Tag in Unthreading

In eJSVM, first-class JavaScript values are represented by the `JSValue` type (Section 2.2). eJSVM is customizable in PTAG value assignments, where the PTAG is the type information in the lowest two or three bits in `JSValue`. For example, it is possible to assign separate PTAG values to simple `Object` and `Array`. By threading, the PTAG value cannot be preserved in a threaded pointer because the LSB, which overlaps with the PTAG, is used as a terminal bit in the header. If the PTAG were written into a threaded pointer, `unthread` could not identify the end of a threaded list correctly. Nevertheless, unthreading must restore the original PTAGs on the `JSValues` that point to the new address.

In eJSVM, correspondences between HTAGs and PTAGs are determined at VM customization time so that the PTAG is uniquely decided according to the HTAG. Therefore, when updating a threaded pointer to a new destination address, the PTAG value to restore is determined based on the HTAG value that resides at the end of the threaded list. Accordingly, the unthreading operation needs two passes over the threaded list: the first to obtain the HTAG value simply by following the list to its end, and the second to update every

Table 1. Execution environments.

	X64	RP
CPU	Core i7-10700	Cortex-A53 (ARMv8) 64-bit SoC
Frequency	2.90 GHz	1.40 GHz
Memory	32 GB	1 GB
OS	Debian 10.7	Raspbian 9.13
GCC	8.3.0 (Debian 8.3.0-6)	6.3.0 20170516 (Raspbian 6.3.0-18 +rpi1+deb9u1)
eJSVM	64-bit configuration	32-bit configuration

threaded pointer with a new destination address by putting the corresponding PTAG into the obtained HTAG.

5.5 Self-Referencing Pointer in a Meta-Object

When using the boundary tag embedding implementation, if a meta-object has a self-referencing pointer inside, the following must be considered. While scanning the meta-object area in the update-forward-reference phase, if such a live meta-object is found, the thread operation for the self-referencing pointer stores a threaded pointer in the meta-object header. As a result, the original value in the meta-object header, in which the boundary tag of the previous meta-object is embedded, is no longer there.

Fortunately, in eJSVM, a meta-object never has a pointer to itself. When Fusuma is applied to systems other than eJSVM, it might be necessary to consider the above case.

6 Evaluation

To evaluate the space efficiency and performance of Fusuma, we ran benchmark programs on the following two environments, whose details are presented in Table 1.

- a desktop computer with an Intel x64 CPU (X64), and
- a Raspberry Pi 3 Model B+ (RP).

Although RP has a 64-bit CPU, Raspbian OS on RP runs on 32-bit; in particular, the address is 32-bit. Thus, for building eJSVM on RP, we used the default 32-bit configuration. We used these rich environments in order to use utility software for the measurement while keeping the heap size of eJSVM small.

Because we were unable to find a suitable JavaScript benchmark suite for embedded systems, we used standard benchmark suites instead. We used seven programs from the AreWeFastYet benchmark [16] and eight programs from the SunSpider benchmark[3], all of which invoked GC. These 15 programs were slightly modified to run on eJSVM. Their program names are listed in Table 2. In addition, we used dht11 [17] and inc-prop programs, which we created. dht11 is based on a program actually used on IoT devices. It repeatedly converts a sequence of bits from a temperature and humidity sensor into numerical temperature and humidity values. inc-prop is a synthetic program that continuously adds new properties of dynamically created names and, hence, creates new hidden classes and reallocates external property arrays of various sizes. The source code is in our technical report [18].

In each of the two execution environments, we prepared three eJSVMs that used different GC algorithms:

- the mark-sweep algorithm, which is the standard GC in eJSVM (MS),
- Fusuma using the naive boundary tag implementation (TC), and
- Fusuma using the boundary tag embedding technique (TCE).

The MS implementation used a first-fit free list allocator without size segregation. A fragment smaller than four words was attached to its adjacent object. In addition to external fragmentations commonly observed in non-moving GCs, it caused internal fragmentations. GC was invoked when the free space of the heap became less than 1/16 of the total heap size to cope with fragmentation in MS and to make a fair comparison of the three.

6.1 Space Efficiency

We investigated the minimum heap size required to run each benchmark program. We hereinafter call this heap size the *lower limit heap size*, or *lower limit* for short. We determined it by repeatedly executing a benchmark program while altering the heap size by 2 KiB. Table 2 presents the lower limit for each benchmark on X64 and RP. A bracketed value means that the execution time exceeded the realistic time when the heap size was less than the bracketed value. We executed the programs for at least twenty minutes before timeout. Note that we will also use the term "lower limit" to denote this bracketed value.

For MS, the lower limit was unclear; a program worked with heap sizes smaller than the heap size where the program failed to run. The reason was that a change in the heap size triggered GC at a different point, which changed the arrangement of objects in the heap. A different arrangement caused a different spatial overhead because of fragmentation. We regarded the minimum heap size such that execution did not fail at any heap size that was equal to or larger than the minimum as the lower limit of the program for MS.

The programs can be classified into two groups. In one group, the lower limits for TC and TCE were substantially smaller than that for MS. A program in this group has a dagger (†) after its name in Table 2. For the programs in this group, the fragmentation that occurred in MS reduced the efficiency of heap utilization. In contrast, the fragmentation was eliminated by the compaction in TC and TCE. Since the severity of fragmentation depends on the program, the ratio of the lower limit of MS to that of TC or TCE differed greatly

[3] https://webkit.org/perf/sunspider/sunspider.html

Table 2. Lower limit of each benchmark program.

Program	X64 (KiB)			RP (KiB)		
	MS	TC	TCE	MS	TC	TCE
AreWeFastYet benchmark						
DeltaBlue†	13,170	(6,440)	(6,440)	9,464	(3,226)	(3,222)
Havlak†	23,090	(10,566)	(10,562)	20,702	(5,294)	(5,294)
CD†	984	584	582	490	298	296
Bounce	50	50	48	32	28	28
Mandelbrot	48	48	48	28	28	26
Sieve†	138	72	70	74	38	38
Storage	550	550	548	278	278	276
SunSpider benchmark						
3d-cube	46	48	46	26	26	26
3d-morph†	(734)	(616)	(616)	(458)	(370)	(370)
access-binary -trees	44	46	44	24	26	24
access-nbody	44	48	44	26	26	26
math-partial -sums	44	46	44	24	26	24
math-spectral -norm	44	46	44	24	26	24
string-base64†	178	96	96	136	88	88
string-fasta	46	48	46	26	26	26
Our programs						
dht11†	88	52	50	48	28	28
inc-prop†	320	116	114	164	58	58

from program to program. Although programs in this group tended to require a large heap, programs requiring a heap of around 100 KiB or smaller also fell into this group. In particular, for dht11, an IoT-oriented program, TCE reduced the lower limit for RP by 20 KiB (42%) compared with MS.

For the other group, serious fragmentation did not occur. Hence, the lower limits for MS and TCE were similar. However, we had to reserve a margin area, which was 1/16 of the heap in this evaluation, for possible fragmentation for MS, while we could reduce the margin for TCE.

The lower limit for TC was larger than that for TCE by 2 KiB for most programs because boundary tags were created for each meta-object. In TCE, the spatial overhead of the boundary tags was reduced.

In summary, we confirmed that TC and TCE reduced the lower limits of programs subject to fragmentation in MS by eliminating the fragmentation by means of the compaction. We also confirmed that the spatial overhead caused by boundary tags, which was observed in TC, was eliminated by the boundary tag embedding technique.

eJSVM is targeted at embedded systems with 100 KiB, and fragmentation may occur even in such systems. In fact, in dht11, the lower limit of MS was larger than that of TC, indicating that fragmentation had a significant impact. TCE is useful in systems targeted by eJSVM, because it is capable of alleviating the space performance degradation caused by fragmentation and by boundary tags.

6.2 Time Efficiency

Figure 13 plots the execution times and GC times of each program against the heap sizes. The figure presents only typical results because of the page limitation. Complete results can be found in the technical report [18]. For each program, the execution times are normalized to the fastest execution time, and the heap sizes are normalized to the lower limit heap size of TCE in Table 2. We executed each program ten times for each heap size and plotted the mean value with quartiles. Missing data points indicate that the execution failed at the heap size.

In general, the execution time comprises the mutator time and the GC time. The mutator time is affected by the following factors.

- TC and TCE may improve locality, which would improve performance for TC and TCE.
- Allocation may be slow in MS because it may follow the free list.
- TC and TCE require an extra shift instruction to remove the terminal bit to access the HTAG because the LSB of the header word is used as the terminal bit, as presented in Section 5.3.

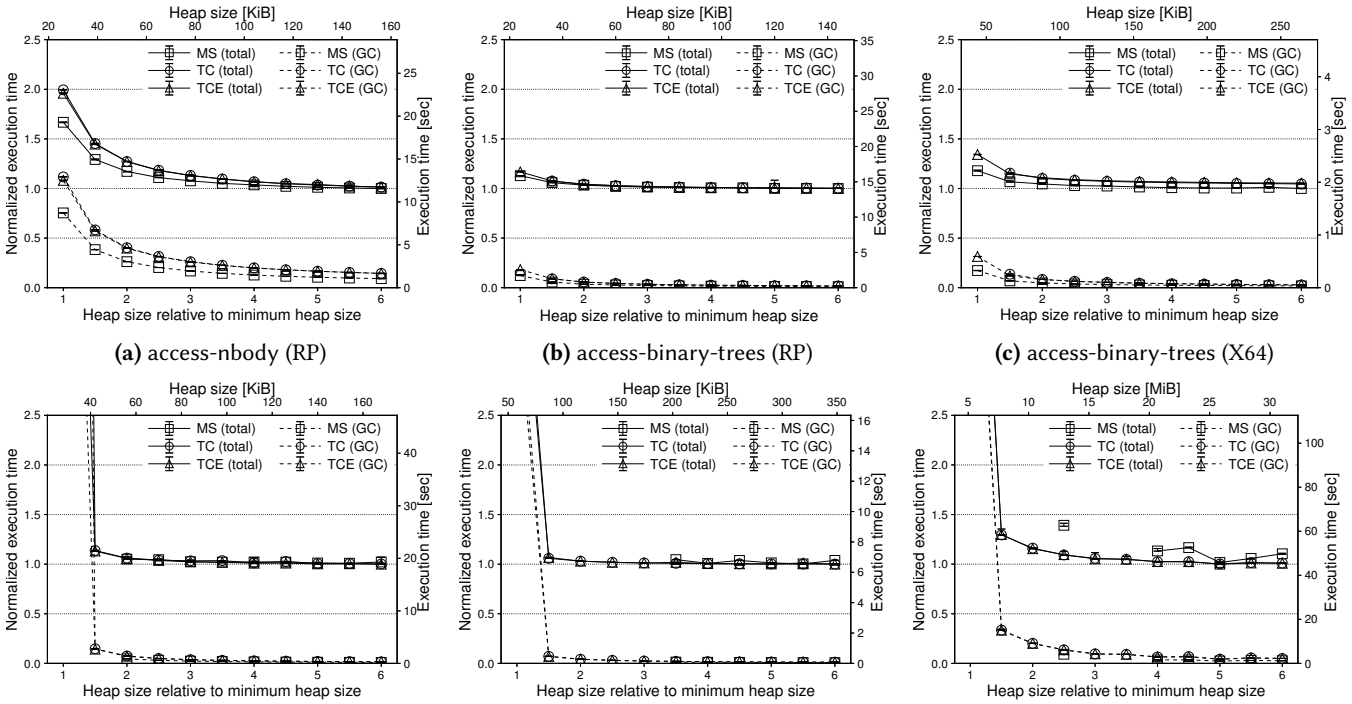
As for the GC time,

- compacting GC tends to be slow.

In fact, we observed that TCE spent at most 2.50× longer for GC than MS, which happened for DeltaBlue at 5.5× the minimum heap size. The effect of each factor depended on the program and the heap size.

access-nbody showed typical behavior of programs in which serious fragmentation did not occur. access-nbody created a bunch of floating point number objects. They were of fixed size and usually ephemeral. With a small heap, MS ran faster than TC and TCE because MS spent a shorter GC time. The GC algorithms had almost the same number of GC cycles (45, 631 cycles for MS, 43, 128 cycles for TC, and 41, 749 cycles for TCE at the minimum heap size), because fragmentation was not serious. However, TC and TCE took longer for a single GC cycle because they needed compaction. For example, TCE took 1.44× longer than MS for GC for the minimum heap size. As the heap became larger, GC operated less frequently, and the difference between MS and the others became smaller. At 6× the minimum heap size, all three GC algorithms showed almost the same performance in terms of total execution time.

TC and TCE might improve locality for access-nbody. The difference between the total time and GC time, i.e., mutator time, was smaller for TC and TCE compared with the difference for MS. For example, at 6× the minimum heap size, the total execution times for all three GC algorithms were almost the same, while the GC time for MS was shorter than TC and TCE. Note that MS always succeeded in allocating a floating point number object from the first chunk on the free list because the floating point number object consisted



(a) access-nbody (RP) **(b)** access-binary-trees (RP) **(c)** access-binary-trees (X64)

(d) dht11 (RP) **(e)** inc-prop (RP) **(f)** Havlak (RP)

Figure 13. Total and GC times of selected benchmark programs.

of two words, which was smaller than the minimum free chunk size of four words.

In contrast, for access-binary-trees, the mutator times for TC and TCE were not significantly shorter than that of MS. In X64, their mutator times were even longer.

dht11, inc-prop, and Havlak caused serious fragmentation. For dht11, MS failed to run at 1.5× the minimum heap size, while TC and TCE ran in a reasonable time. TC and TCE ran at the minimum heap size, though they did so very slowly (33.5× and 10.4× the fastest execution for TC and TCE), performing GC frequently.

Fragmentation was more serious for inc-prop. MS failed to run at 3× the minimum heap size. Fragmentation in this case came from reallocation of the arrays of the property names in hidden classes, which were meta-objects, and the external property arrays of ordinary objects. inc-prop continuously added properties with new names. This behavior tended to cause serious fragmentation. In eJSVM, a hidden class manages the set of property names with an array. Thus, adding a new property caused reallocation of the array in order to expand it. Furthermore, the object to which a property was added reallocated its external property array. Because properties were added one by one, the sizes of the reallocated arrays increased little by little. Thus, objects with a variety of sizes were created, causing serious fragmentation. For TC and TCE, any fragmentation caused by ordinary objects and meta-objects was successfully eliminated.

For Havlak, MS showed a chaotic behavior. As shown in Table 2, the lower limit of heap size for MS was 20,702 KiB in RP. Thus, it failed to execute at 4× the minimum heap size (18,529 KiB). However, it completed its execution at 3× the minimum heap size. Furthermore, for larger heaps, the curve for MS was not monotonic, bouncing on the curves for TC and TCE. Given that MS spent a similar or shorter GC time than the others, the extra overhead applied to the mutator time varied from one heap size to another. The reason might be that how seriously the heap became fragmented varied. Generally speaking, when a heap is fragmented, allocation tends to follow more chunks on the free list. For a free list allocator, how seriously the heap becomes fragmented depends on the point when GC is triggered. For example, performing GC at the moment when all recently allocated objects become unreachable does not cause fragmentation while performing GC after the next object is allocated results in the heap being divided into two parts by the object.

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 the double-ended method because most of the GC time was spent on scanning the ordinary object area. To confirm that this was the case, we prepared two separate heaps, one for ordinary objects and the other for meta-objects, and implemented two eJSVMs that used the following GC strategies.

- GC that targeted only the heap for ordinary objects by using Jonkers's threaded compaction. This did not reclaim garbage in the heap for meta-objects (VM1).
- GC that targeted both heaps by using almost the same algorithm as Fusuma (VM2).

We ran benchmark programs on both eJSVMs, setting the same heap sizes to make the timings of GC invocation the same, and compared them by calculating the ratio of the GC time of VM1 to that of VM2 for each program. The results showed that the minimum ratio was 0.88 on X64 and 0.87 on RP, with averages of 0.96 and 0.95 respectively. These results indicated that compaction of ordinary object area was the dominant factor of the GC time.

7 Related Work

In statically typed languages such as Java, it is common to hold the layout of an object in a meta-object. Such an implementation suffers from the same problem that has been focused on in this paper.

MMTk [3], which is a memory management system for the Jikes RVM [1], is a framework for implementing various GC algorithms. MMTk allocates meta-objects in a dedicated area managed by the mark-sweep GC, which makes it possible to implement a GC that moves ordinary objects. In comparison, Fusuma makes it possible to manage both ordinary objects and meta-objects in the same heap area without worrying about exhausting one of the areas.

In compaction algorithms where the source and destination regions of objects do not overlap, meta-objects can be easily referred to during compaction. For example, copying GC [5] simply follows the forwarding pointer of a meta-object when the meta-object has already been copied. If the heap is divided into fixed-length blocks and objects are moved from some blocks to others to create contiguous free space, such as the mixed strategy collection [15] or G1GC [8], meta-objects can be referred to during GC in the same way as in copying GC.

To overlap between source and destination regions for objects, GC requires two distinct regions for objects to be moved. Specifically, semi-space copying GC requires a heap twice as large as that required by the total amount of objects used by the program. In contrast, Fusuma has no spatial overhead.

Often in concurrent GC, where mutators and collectors operate concurrently, meta-objects are referred to during GC. Even in concurrent GC with compaction [2, 8, 21], most algorithms ensure that the source and destination regions for objects do not overlap.

Sliding compaction algorithms other than threaded compaction include Lisp2 compaction [14] and Break-Table compaction [10]. As in threaded compaction, it is not easy for these algorithms to compact ordinary objects and meta-objects at the same time. In addition, Lisp2 compaction has the disadvantage that every object needs an extra word for the forwarding pointer.

Lisp2 compaction [14] first determines live objects and then performs the destination determination phase. In this phase, the heap area is scanned from the left. Every time a live object is found, the destination address where the object is to be moved is written into the forwarding pointer area. Next, the pointer update phase is performed. In this phase, pointers in each object are updated so that they point to the new destination address. At this time, information on the object layout is required. However, it is impossible to know the object layout because pointers held in meta-objects may have been updated. This problem can be solved by updating ordinary objects pointers before updating the pointers held in meta-objects in the way described in this paper.

Break-Table compaction [10] manages object destinations in a table called the break table, which records the address of each live object and the amount of movement of the object. The break table is divided into several fragments, which are created in gaps between live objects and are collected in one place as the objects are moved. Although there is no spatial overhead, the computation time is $O(N \log N)$, where N is the number of objects, because the collected fragments are sorted. In addition, until all objects are moved, the break table is not sorted, and consequently the destination of an object is not determined. Therefore, the object layout cannot be referred to if meta-objects are targets of compaction.

A mark-sweep-compact collector [19] usually performs mark-sweep GC while occasionally compacting the heap. This can reduce the amortized cost of compaction. Although this technique is orthogonal to the present proposal, it is listed under future work.

8 Conclusion

This paper has proposed Fusuma, a double-ended threaded compaction that allows ordinary objects and meta-objects to be allocated in the same heap. By using the boundary tag embedding technique, the proposed compaction can be implemented without any extra space for each object.

We implemented Fusuma in eJSVM and confirmed to improve space efficiency compared with mark-sweep GC. The GC overhead of Fusuma was less than 2.50× that of mark-sweep GC for a realistic heap size.

Acknowledgments

We are grateful to Richard Jones and Stefan Marr of the University of Kent for providing us with useful comments. We are also grateful for the support of the JSPS through KAKENHI grant number 18KK0315.

References

[1] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Syst. J.* 44, 2 (2005), 399–418. https://doi.org/10.1147/sj.442.0399

[2] David F. Bacon, Perry Cheng, and V. T. Rajan. 2003. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for Java. In *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2003)*. ACM, 81–92. https://doi.org/10.1145/780732.780744

[3] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *26th International Conference on Software Engineering (ICSE 2004)*. IEEE Computer Society, 137–146. https://doi.org/10.1109/ICSE.2004.1317436

[4] Craig Chambers, David M. Ungar, and Elgin Lee. 1989. An Efficient Implementation of SELF – a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Conference Proceedings on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA 1989)*. ACM, 49–70. https://doi.org/10.1145/74877.74884

[5] Chris J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. *Commun. ACM* 13, 11 (1970), 677–678. https://doi.org/10.1145/362790.362798

[6] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento mori: dynamic allocation-site-based optimizations. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management (ISMM 2015)*. ACM, 105–117. https://doi.org/10.1145/2754169.2754181

[7] Ulan Degenbaev, Michael Lippautz, and Hannes Payer. 2019. Concurrent marking of shape-changing objects. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM 2019)*. ACM, 89–102. https://doi.org/10.1145/3315573.3329978

[8] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM 2004)*. ACM, 37–48. https://doi.org/10.1145/1029873.1029879

[9] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1984)*. ACM, 297–302. https://doi.org/10.1145/800017.800542

[10] Bruce K. Haddon and William M. Waite. 1967. A Compaction Procedure for Variable-Length Storage Elements. *Comput. J.* 10, 2 (1967), 162–165. https://doi.org/10.1093/comjnl/10.2.162

[11] Urs Hölzle, Craig Chambers, and David M. Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP'91 European Conference on Object-Oriented Programming (Lecture Notes in Computer Science, Vol. 512)*. Springer, 21–38. https://doi.org/10.1007/BFb0057013

[12] H. B. M. Jonkers. 1979. A Fast Garbage Compaction Algorithm. *Inf. Process. Lett.* 9, 1 (1979), 26–30. https://doi.org/10.1016/0020-0190(79)90103-0

[13] Haim Kermany and Erez Petrank. 2006. The Compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*. ACM, 354–363. https://doi.org/10.1145/1133981.1134023

[14] Donald Ervin Knuth. 1997. *The art of computer programming, Volume I: Fundamental Algorithms, 3rd Edition*. Addison-Wesley. https://www.worldcat.org/oclc/312910844

[15] Bernard Lang and Francis Dupont. 1987. Incremental incrementally compacting garbage collection. In *Proceedings of the Symposium on Interpreters and Interpretive Techniques*. ACM, 253–263. https://doi.org/10.1145/29650.29677

[16] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS 2016)*. ACM, 120–131. https://doi.org/10.1145/2989225.2989232

[17] Hiro Onozawa, Hideya Iwasaki, and Tomoharu Ugawa. 2021. Customizing JavaScript Virtual Machines for Specific Applications and Execution Environments. *Computer Software* 38, 3 (2021). to appear (in Japanese).

[18] Hiro Onozawa, Tomoharu Ugawa, and Hideya Iwasaki. 2021. Fusuma: Double-Ended Threaded Compaction (Full Version). https://ipl.cs.uec.ac.jp/~iwasaki/eJS/technical-report/Fusuma.pdf

[19] Tony Printezis. 2001. Hot-Swapping Between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*. USENIX. http://www.usenix.org/publications/library/proceedings/jvm01/printezis.html

[20] Tomoharu Ugawa, Hideya Iwasaki, and Takafumi Kataoka. 2019. eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems. *J. Comput. Lang.* 51 (2019), 261–279. https://doi.org/10.1016/j.cola.2019.01.003

[21] Tomoharu Ugawa, Carl G. Ritson, and Richard E. Jones. 2018. Transactional Sapphire: Lessons in High-Performance, On-the-fly Garbage Collection. *ACM Trans. Program. Lang. Syst.* 40, 4 (2018), 15:1–15:56. https://doi.org/10.1145/3226225

A inc-prop Benchmark

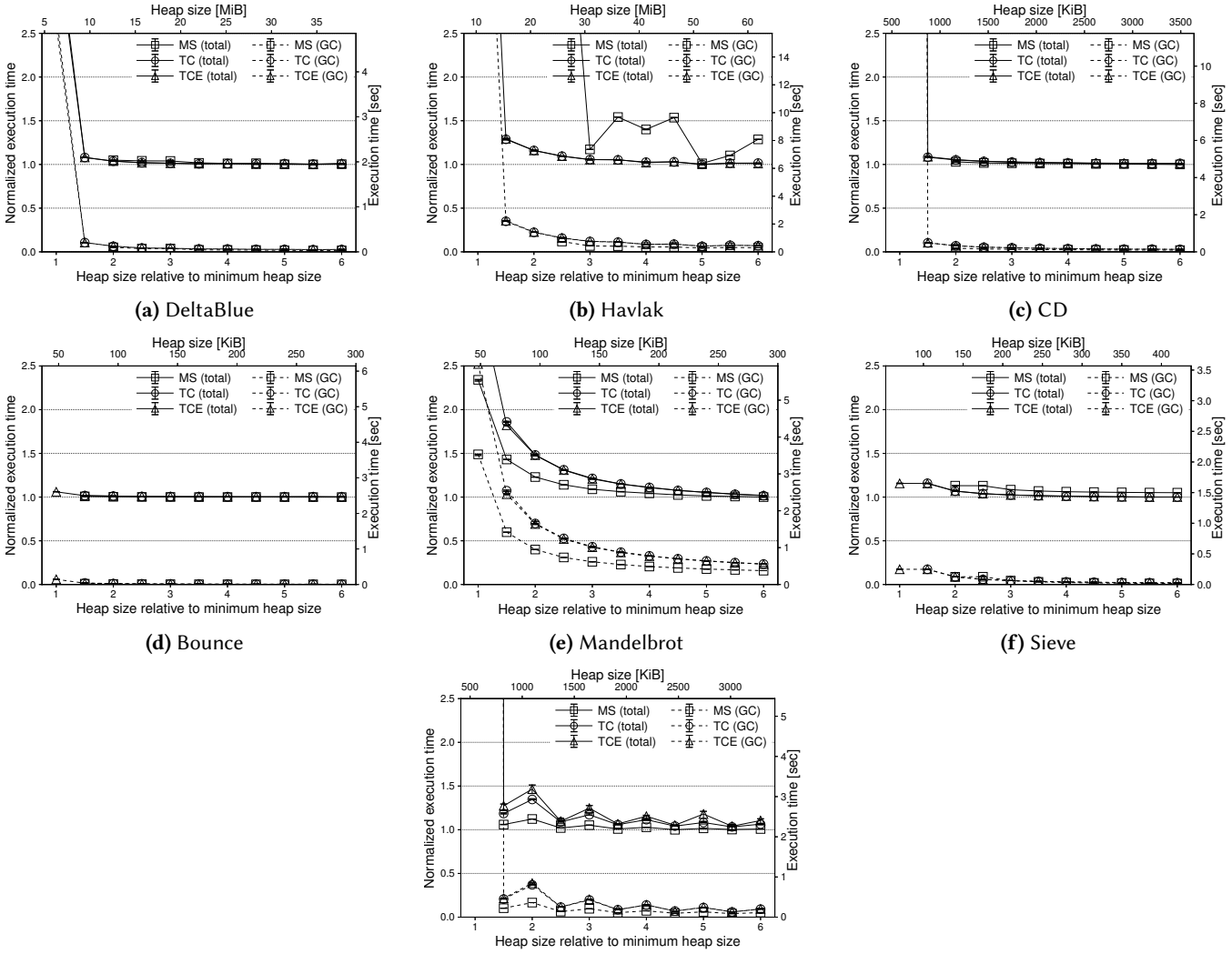
The synthetic benchmark program inc-prop is shown in figure 14.

B Benchmark Results

Figures 15 to 18 plot the execution time and the GC time against the heap size for all benchmarks.

```
function f(iter, loop) {
  var base = [
    "a","b","c","d","e","f","g","h","i",
    "j","k","l","m","n","o","p","q","r",
    "s","t","u","v","w","x","y","z"];
  var props = [];
  var objs = [];
  iter = iter * 26;
  for (var i = 0; i < iter; ++i) {
    var size = 3 + Math.floor(iter / 26);
    for (var s = 0; s < size; ++s) {
      props[s] = base[(i + s) % 26];
    }
    for (var j = 0; j < loop; ++j) {
       var o = {};
       for (var s = 0; s < size; ++s)
         o[props[s]] = 0;
       objs[Math.floor(i / 2)] = o;
    }
  }
  return objs;
}

measure(function() { f(32, 512); });
```

Figure 14. inc-prop benchmark.

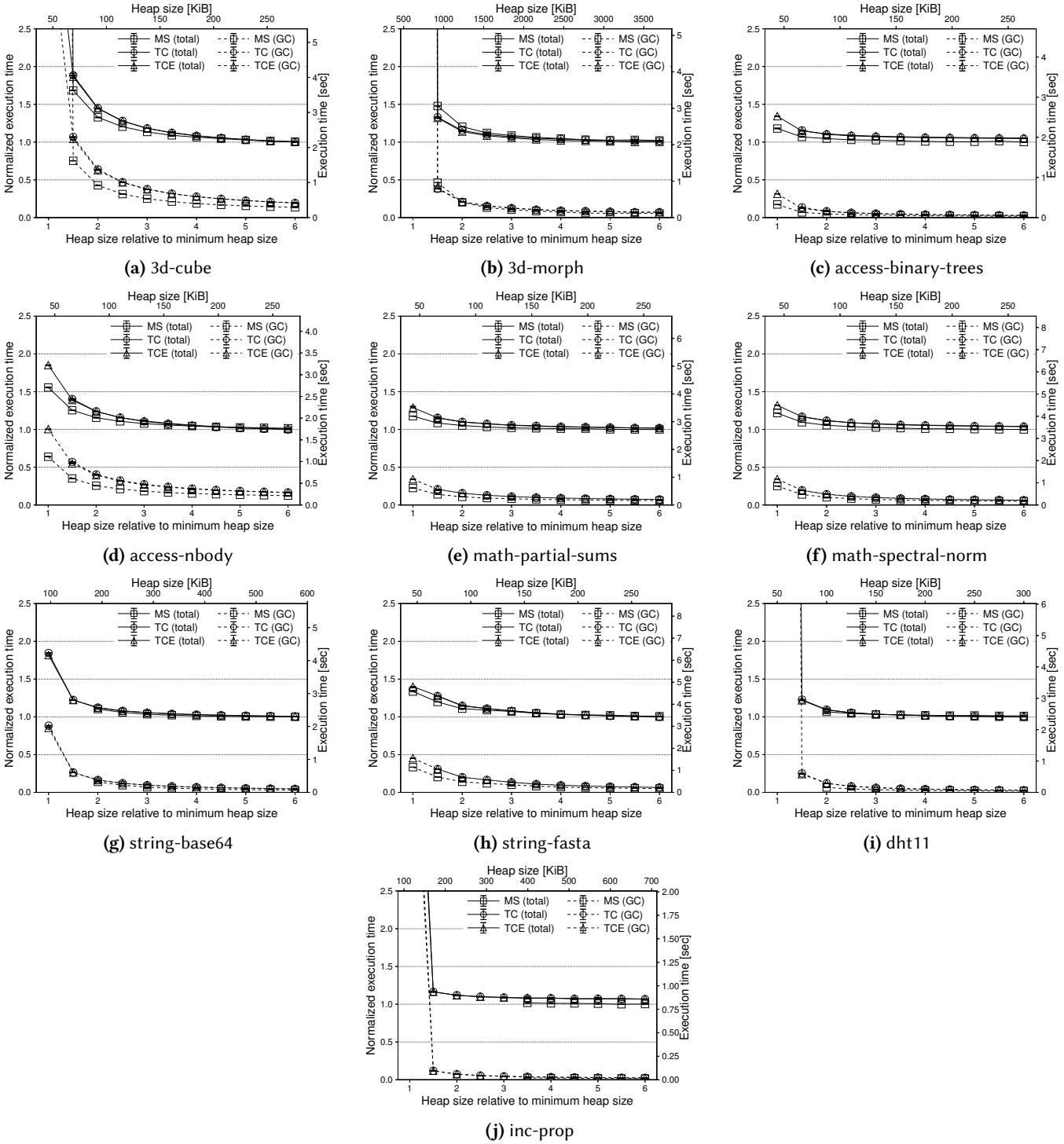


(a) DeltaBlue **(b)** Havlak **(c)** CD

(d) Bounce **(e)** Mandelbrot **(f)** Sieve

(g) Storage

Figure 15. Execution time and GC time vs. heap size for X64 (1).



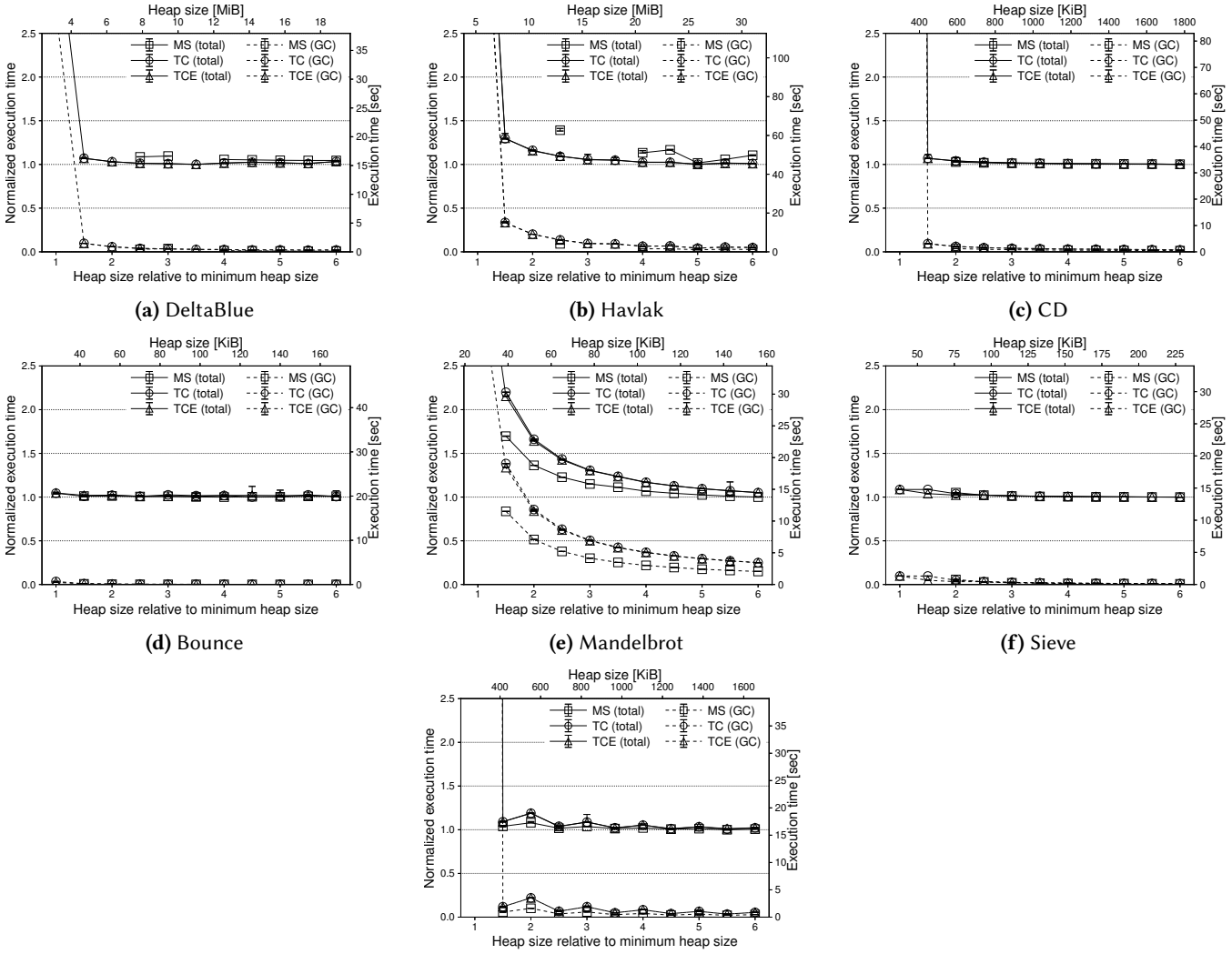
(a) 3d-cube **(b)** 3d-morph **(c)** access-binary-trees

(d) access-nbody **(e)** math-partial-sums **(f)** math-spectral-norm

(g) string-base64 **(h)** string-fasta **(i)** dht11

(j) inc-prop

Figure 16. Execution time and GC time vs. heap size for X64 (2).

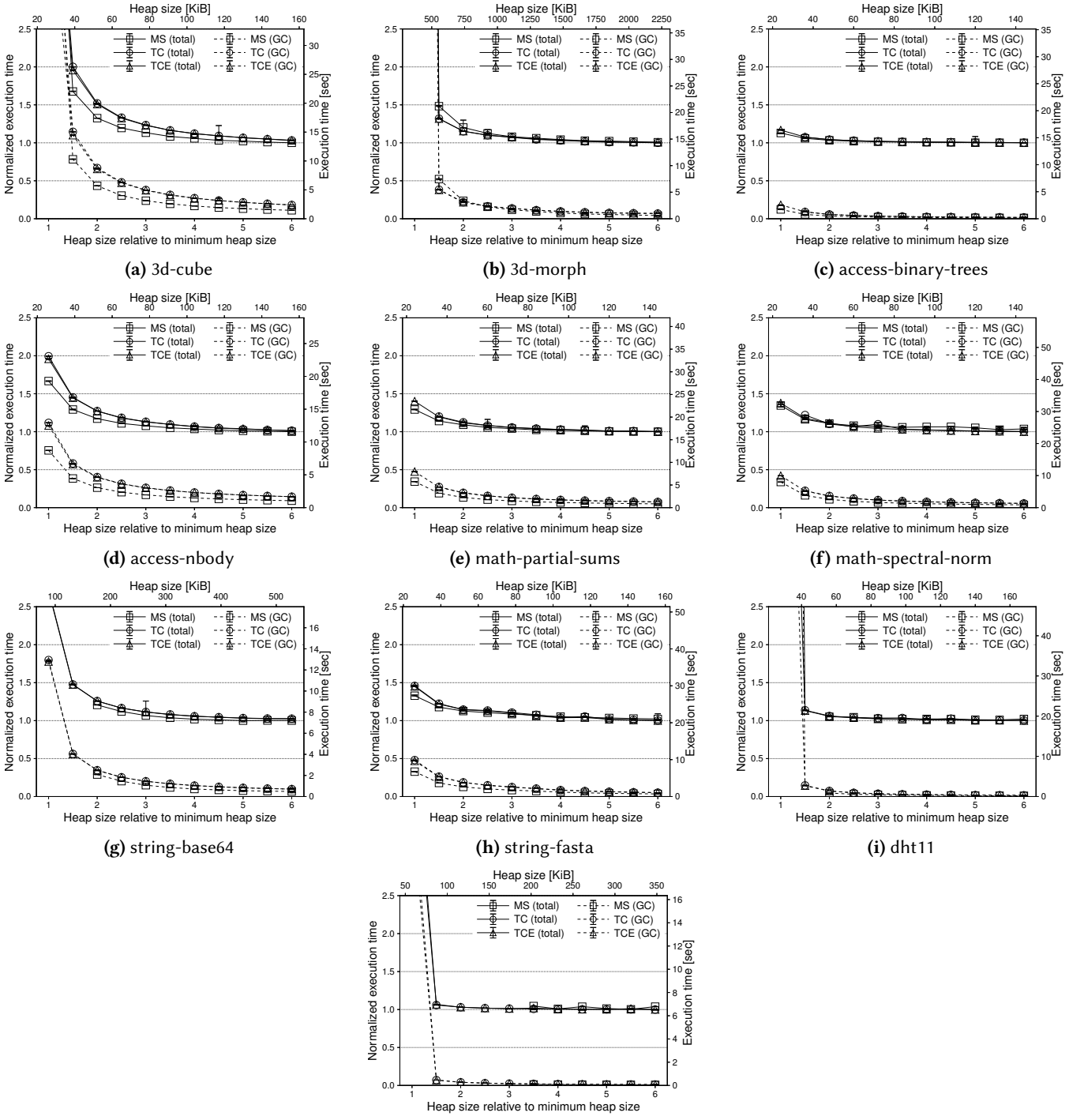


(a) DeltaBlue **(b)** Havlak **(c)** CD

(d) Bounce **(e)** Mandelbrot **(f)** Sieve

(g) Storage

Figure 17. Execution time and GC time vs. heap size for RP (1).



(a) 3d-cube **(b)** 3d-morph **(c)** access-binary-trees

(d) access-nbody **(e)** math-partial-sums **(f)** math-spectral-norm

(g) string-base64 **(h)** string-fasta **(i)** dht11

(j) inc-prop

Figure 18. Execution time and GC time vs. heap size for RP (2).