修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　　情報理工学研究科　情報・ネットワーク工学専攻　博士前期課程		
氏　　名	赤澤亮弥	学籍番号	1631003
論 文 題 目	言語仮想機械におけるカスタマイズ可能なごみ集めモジュールの実装		

要　旨

近年では，あらゆるモノをネットワークで結ぶ新たな技術として IoT（Internet of Things）に注目が集まっている．IoT を始めとして，組込み機器上で動作させるプログラムはハードウェアに最適化させるために，C やアセンブラといった低レベルのプログラミング言語を用いて開発されることが多い．しかし，こうした言語による開発には，時間がかかる上に，バグが発生しやすいという問題がある．

この問題を解決するため，組込み向けプログラムの開発に JavaScript を利用することを目的とし，各 IoT デバイスとその上で動作するプログラムに特化した JavaScript 仮想機械を生成するカスタマイズ可能なフレームワークが開発されている．このフレームワークは，プログラマがデータ型に関する定義ファイルを作成することで，デバイスや動作させるプログラムに最適化した仮想機械を生成できる．

本研究ではこのフレームワークのカスタマイズ性の拡張を目的とし，メモリ管理の部分に着目した．このフレームワークに現在実装されているごみ集めは，マークスイープ方式と呼ばれるアルゴリズムのものだが，本研究では新たにコピー方式と呼ばれるアルゴリズムも実装し，プログラマが，動作させるプログラムに合わせて使用するごみ集めのアルゴリズムを選択できるようにした．また，性能評価実験を行い，ごみ集めのアルゴリズムをプログラムごとに選択できることの効果を確認した．

平成 29 年度修士論文

言語仮想機械における
カスタマイズ可能な
ごみ集めモジュールの実装

電気通信大学 大学院 情報理工学研究科
情報・ネットワーク工学専攻
コンピュータサイエンスコース

学籍番号　：1631003
氏名　：赤澤 亮弥
主任指導教員　：岩崎 英哉 教授
指導教員　：寺田 実 准教授
提出日　：2017 年 1 月 29 日

要旨

近年では，あらゆるモノをネットワークで結ぶ新たな技術として IoT（Internet of Things）に注目が集まっている．IoT を始めとして，組込み機器上で動作させるプログラムはハードウェアに最適化させるために，C やアセンブラといった低レベルのプログラミング言語を用いて開発されることが多い．しかし，こうした言語による開発には，時間がかかる上に，バグが発生しやすいという問題がある．

この問題を解決するため，組込み向けプログラムの開発に JavaScript を利用することを目的とし，各 IoT デバイスとその上で動作するプログラムに特化した JavaScript 仮想機械を生成するカスタマイズ可能なフレームワークが開発されている．このフレームワークは，プログラマがデータ型に関する定義ファイルを作成することで，デバイスや動作させるプログラムに最適化した仮想機械を生成できる．

本研究ではこのフレームワークのカスタマイズ性の拡張を目的とし，メモリ管理の部分に着目した．このフレームワークに現在実装されているごみ集めは，マークスイープ方式と呼ばれるアルゴリズムのものだが，本研究ではコピー方式と呼ばれるアルゴリズムも実装し，仮想機械が使用するごみ集めのアルゴリズムを，動作させるプログラムに合わせて，仮想機械の生成時にプログラマが選択できるようにした．また，性能評価実験を行い，ごみ集めのアルゴリズムをプログラムごとに選択できることの効果を確認した．

目次

1 序論

1.1 背景

現在，センサやデバイスなどの「モノ」を，通信を介してクラウドやサーバ等に繋いで相互に制御する仕組み IoT（Internet of Things）に注目が集まっている [6].

一般的に IoT デバイス上で動作させるプログラムの開発には，ハードウェアに最適化させるため，C やアセンブリ言語といった低レベルのプログラミング言語が用いられることが多い．しかし，こうした言語を用いてプログラムを開発するとき，ハードウェアに最適化させるために細かいパラメータも全てプログラマが指定する必要がある．さらに，ソースコードを見ても処理の内容を把握しづらく，プログラムミスに繋がりやすい．そのため，開発には多大な時間がかかる傾向がある．また，それらに加えて，プログラムの移植性に欠けるという問題もある．

現在，そうした問題を解決する手段として，高知工科大学の鵜川研究室と電気通信大学の岩崎研究室が合同で，embedded JavaScript（eJS） [?] という，JavaScript をベースとした言語処理系を開発している．JavaScript は Web 開発の領域で広く用いられているプログラミング言語であるが，スクリプト言語であるため，コードを書いてからプログラムを実行するまでの時間が短く，プログラムの試作が容易であるという利点がある．また，JavaScript のイベント駆動型プログラミングモデルは，IoT プログラムのようにセンサや通信を用いるプログラムに適していると考えられる [2, 10].

しかし，JavaScript で開発する場合，プログラムについては低レベル言語ほどの最適化が望めない．そこで，eJS では IoT デバイスとその上で動作させるプログラムの性質に着目した仮想機械の最適化を試みている．IoT デバイスは一般に CPU やメモリ，バッテリー等について，必要最低限の性能しか持たない一方，一般の PC と異なり，動作させるプログラムは事前に決まっているという特徴がある．eJS では，eJSToolKit（eJSTK）と呼ばれるフレームワークから eJS 仮想機械（eJSVM）を生成することができ，プログラマは eJSTK に定義ファイルを与えることで，生成される eJSVM で使用するデータ型を選択することができる．プログラマが eJSVM を，デバイスやその上で動作させるプログラムに最適な形にカスタマイズすることで，IoT デバイスの限られた性能でも優れたパフォーマンスを発揮できる．

1.2 目的

本研究では，eJSTK のさらなるカスタマイズ要素として，ごみ集め（Garbage Collection, GC）アルゴリズムの選択機構を実現する．GC とは，プログラムが動的に確保した領域のうち，不要になった領域を自動的に回収する機構である．JavaScript の処理系においては，ランタイム

に GC が含まれている．

GC には様々なアルゴリズムが提案されている．それらはそれぞれ一長一短の性質を持ち，あらゆる面に優れた完璧なアルゴリズムは存在しない．eJSTK に実装されている GC アルゴリズムはマークスイープ方式と呼ばれる一種類だけである．しかし，動作させるプログラムごとに，適した GC アルゴリズムは異なると考えられるため，本研究では新たにコピー方式アルゴリズムを eJSTK に実装し，プログラマが，生成される eJSVM で使用される GC を 2 種類から選択できるようにした．

1.3 本論文の構成

本論文は，2 章で eJS について詳しく説明し，3 章では eJSVM における GC について述べる．4 章でコピー方式 GC の実装について述べ，5 章では，実装したマークスイープ方式 GC とコピー方式 GC の性能を比較する実験について述べ，性能を評価する．6 章で関連研究について述べ，最後の 7 章で本論文をまとめる．

2　eJS

この章では，高知工科大学の鵜川研究室と電気通信大学の岩崎研究室が合同で開発している，embedded JavaScript（eJS） [?] について説明する．

2.1　概要

eJS 処理系の構成を，図 2.1 に示す．eJS Compiler（eJSC）は，JavaScript で記述されたソースプログラムを受け取ると，それを解析し，VM 命令列を生成する．このとき使用される VM 命令セットは，eJS 処理系で規定された独自のものである．eJSTK は eJSVM のソースプログラムを含むフレームワークであり，ユーザによるカスタマイズの定義に対応して，カスタマイズされた eJSVM を生成することができる．生成された eJSVM をデバイス上に配置し，eJSC が生成した VM 命令列を eJSVM にロードすることで，プログラムを実行することができる．

eJSTK の構成を図 2.2 に示す．eJSTK が提供する C 言語で記述された既定の処理系のソースプログラムと，プログラマが記述した定義ファイルを元に生成されるソースプログラムから，カスタマイズされた eJSVM を生成する．

2.2　eJSC

eJSC は JavaScript で記述されたソースプログラムを受け取り，eJS 処理系における独自の VM 命令列を生成する．eJSC がサポートするのは ECMAScript 5.1 のサブセットであるが，eJS が対象とする IoT デバイスは性能が限られているため，いくつかの機能がサポートされない．

まず，eval 関数をサポートしない．eval 関数は与えられた文字列を JavaScript コードとして評価する関数であるが，図 2.1 に示した通り，eJS の処理系においては評価に必要となるコンパイラがデバイス上に無いためである．

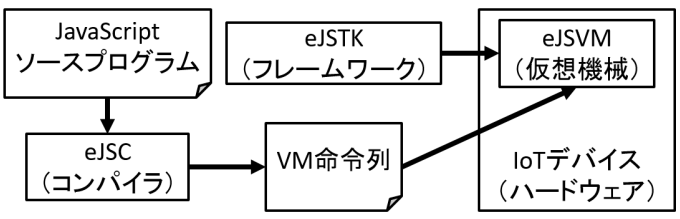

図 2.1　eJS 処理系の構成

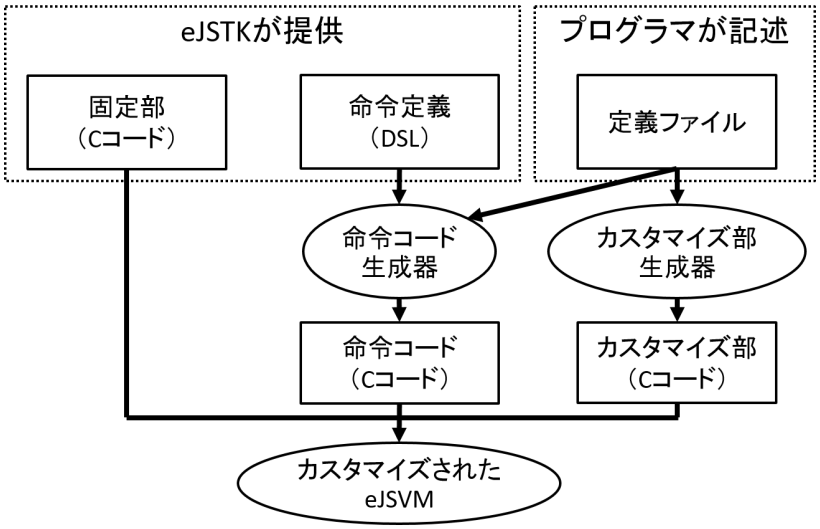

図 2.2　eJSTK の構成

また，プログラムミスに繋がりやすい以下の機能をサポートしていない．

- with 文
- delete 文
- グローバル変数宣言時の var の省略

with 文は性能の低下や曖昧性から公式で非推奨となっている．delete 文を使用するとオブジェクトの持つプロパティが把握しづらくなることから，また，var を省略した変数はグローバル変数として宣言されるがローカル変数に誤解しやすいことから，それぞれプログラムミスに繋がりやすい．

最後に，他のコードで代替できる以下の機能がサポートされない．

- switch 文
- 配列とオブジェクトの初期値リテラル
- 名前付きの関数定義

switch 文は if 文で代用できる．初期値リテラルは，後から値を設定することで同じ処理を行える．名前付きの関数は，無名関数を定義してから新たに宣言した変数に束縛することで実現できる．

2.3 eJSVM

先述した通り，eJSVM は定義ファイルを元にしてカスタマイズされるが，この節ではカスタマイズの内容によらない基本的なデータ表現やメモリ管理について述べる．初めに，eJSVM において用いられるいくつかの用語を以下に示す．

JS 型 JavaScript の仕様にある 6 種類の型のうち Object 型以外の 5 種類と，Object 型に分類される 7 種類のオブジェクト型を合わせた 12 種類の型の総称．

VM 型 JS 型を表現するために eJSVM において定義された型．

VM 型内部表現 eJSVM における VM 型の内部表現．

オブジェクト eJSVM 自体が内部に確保するヒープメモリ上に割り当てられるデータ．JS オブジェクト非 JS オブジェクトの 2 種類が存在する．

JS オブジェクト VM 型のオブジェクト．

非 JS オブジェクト 関数フレームなど，JS オブジェクト以外の，eJSVM 自体が動作するのに必要なデータ構造のオブジェクト．

2.3.1 データ型

表 2.1 に JavaScript の仕様にあるデータの型を示す．JavaScript の仕様において，型は 6 種類存在する．その中で Object 型はさらに 7 種類のオブジェクトに分類される．eJS の処理系においては，これらの型及びオブジェクトの分類をまとめて“JS 型”と呼ぶ．さらに，JS 型を eJSVM において表す型として“VM 型”が定義されている．

JS 型に対応する VM 型を表 2.2 に示す．Undefined など，属する値が定数個しかない型はまとめて Special という VM 型で定義される．また，0 や 1 などの単純な整数値を常に浮動小数点で表現するのは無駄が多いため，JS 型の Number は VM 型において Fixnum と Flonum に分けられている．

eJSVM では，64 ビット符号なし整数型である uintptr_t 型のエイリアスとして JSValue という型が定義されており，VM 型はこの JSValue を通して扱う．JSValue には，扱う VM 型に応じて，即値や実体データへのポインタが格納される．VM 型の実体データは，eJSVM が内部に確保するヒープメモリ上に割り当てられ，この実体データを以下では“オブジェクト”と呼ぶ．特に，VM 型のオブジェクトのことは，2.3.2 節で述べるそれ以外のオブジェクトと区別するために“JS オブジェクト”と呼ぶことにする．本論文で用いられる“オブジェクト”という語が示すものは，JavaScript の言語仕様にある Object とは異なるものであることに注意されたい．

eJSVM において，オブジェクトは 8 バイトアラインメント，すなわち，オブジェクトが 8 の倍数のアドレスに割り当てられるように実装されているため，オブジェクトの先頭アドレスの下

表 2.1　JavaScript におけるデータ型

型	Object の分類	JS 型	表すもの
Undefined		Undefined	undefined
Null		Null	null
Boolean		Boolean	true と false の 2 値
String		String	UTF-16 文字列
Number		Number	数値
Object	Object Object	SimpleObject	通常のオブジェクト
	Array	Array	配列
	Function	Function	関数
	Regexp	Regexp	正規表現
	Boolean Object	BooleanObject	Boxing された Boolean オブジェクト
	String Object	StringObject	Boxing された String オブジェクト
	Number Object	NumberObject	Boxing された Number オブジェクト

表 2.2　JS 型と VM 型の対応

JS 型	VM 型	表すもの
Undefined	special	
Null	special	
Boolean	special	
String	string	文字列
Number	fixnum	61-bit 符号付整数
	flonum	C の double 型
SimpleObject	simple_object	通常のオブジェクト
Array	array	配列
Function	function	ユーザ定義関数
	builtin	組込み関数
Regexp	regexp	正規表現

位 3 ビットは必ず“000”となる．そこで，この下位 3 ビットをそのデータの VM 型を判別するためのポインタタグとして利用する（図 2.3）．ポインタタグによって JSValue の下位 3 ビットが使われるため，数値などは上位 61 ビットに格納する．各 VM 型を表すポインタタグは表 2.3 の通りである．なお，厳密には，object という名前の VM 型は存在しないが，simple_object や array のポインタタグは全て同じ“000”であるため，ここでは便宜上，JavaScript の型における Object に相当するこれらの VM 型を“object 型”と総称している．

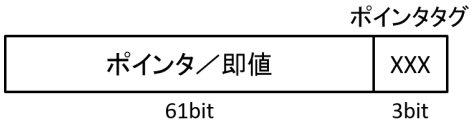

図 2.3　JSValue

表 2.3　ポインタタグ

タグ	VM 型	ポインタ／即値
000	object	ポインタ
001	空き	–
010	空き	–
011	空き	–
100	string	ポインタ
101	flonum	ポインタ
110	special	即値
111	fixnum	即値

次に VM 型ごとの，JSValue に格納されるデータについて詳しく述べる．

object

実体である JS オブジェクトのアドレスがそのまま JSValue となる．先述の通りオブジェクトは 8 バイトアラインメントされているため，アドレスの下位 3 ビットは“000”となり，そのまま object であることを示すポインタタグとして扱われる．

string

文字列の実体である string オブジェクトのアドレスにポインタタグを付加した値が JSValue となる．

flonum

数値データの実体である flonum オブジェクト）のアドレスにポインタタグを付加した値が JSValue となる．

special

上位 32 ビットを 0 とし，下位 32 ビットからポインタタグの 3 ビットを除いた残りの 29 ビットで値を表す．表現する値を表 2.4 に示す．

表 2.4　special が表す値

JSValue	表す値
0	null
1	undefined
2	false
3	true

図 2.4　ヘッダタグの構造

fixnum

上位 61 ビットが，61 ビット符号付き整数の 2 の補数表現として解釈した整数を表し，下位 3 ビットにポインタタグを保持する．

2.3.2　オブジェクト

eJSVM においてオブジェクトは 2 種類に分けられる．片方は 2.3.1 節で説明した，JS オブジェクトのことで，VM 型のオブジェクトが該当する．もう一方は，それ以外のオブジェクトであり，これを以下では“非 JS オブジェクト”と呼ぶ．非 JS オブジェクトには関数フレームや，JS オブジェクトが持つプロパティのハッシュ表などを表すデータ構造の実体データが該当する．eJSVM は最初に“JS スペース”という名前のヒープ領域を確保し，オブジェクトはこの領域に配置して管理する．

オブジェクトは先頭に 1 ワード分のオブジェクトヘッダを持つ（図 2.4）．ヘッダの下位 8 ビットは，そのオブジェクトがどのデータ構造の実体なのかを示すヘッダタグとして扱われる．上位 32 ビットにヘッダを含めたオブジェクトのワード単位のサイズを保持し，中間の 24 ビットは GC に必要な情報を保持するのに用いられる．GC によって必要な情報の種類や数は異なるため，使用されないビットも存在する．表 2.5 にオブジェクトの種類と対応するオブジェクトタグを示す．表の 2 列目は，JS オブジェクトと非 JS オブジェクトのどちらであるかを表す．なお，JSValue がポインタとしてオブジェクトのアドレスを値に持つ際，アクセスのしやすさを重視し，ヘッダの先頭アドレスではなく，データ本体の先頭アドレスを持つ（図 2.5）．

また，JS オブジェクトのうち，object 型のオブジェクトは共通部分として，図 2.6 に示すよ

うに，ヘッダから続く先頭 4 ワードにプロパティに関する情報として以下のデータを持つ．

- プロパティ数
- プロパティ配列のサイズ
- プロパティ名からプロパティ配列のインデックスへのハッシュ（タグ 20 番の非 JS オブジェクトへのポインタ）
- プロパティ配列（タグ 17 番の非 JS オブジェクトへのポインタ）

表 2.5　オブジェクトの種類

タグ	JS or 非 JS	オブジェクトの種類
4	–	string オブジェクト
5	–	flonum オブジェクト
6	JS	object オブジェクト
7	JS	array オブジェクト
8	JS	クロージャの関数オブジェクト
9	JS	組込み関数の関数オブジェクト
11	JS	正規表現オブジェクト
17	非 JS	object が持つプロパティ
18	非 JS	array オブジェクトが持つ配列
19	非 JS	関数フレーム
20	非 JS	プロパティのハッシュ表
21	非 JS	文字列のキャッシュリスト
22	非 JS	隠れクラス
23	非 JS	JS オブジェクトを扱う配列
24	非 JS	ハッシュイテレータ
25	非 JS	命令コードの配列

2.4　eJSTK

eJSTK は eJSVM を生成するフレームワークであり，プログラマが定義ファイルを記述することで，eJSVM をカスタマイズすることができる．現状の eJSVM のカスタマイズの要素はデータ型に関するものだけだが，本研究ではさらにメモリ管理に関してもカスタマイズできるようにした．

2.4.1　データ型のカスタマイズ

生成される eJSVM で使用するデータ型やポインタタグ，オブジェクトタグは，プログラマが任意に定義することもできる．定義ファイルの一部をリスト 2.1 に示す．

1–9 行目は，string 型や flonum 型といった JS オブジェクトに対し，`normal_string` や `normal_flonum` 等，内部表現におけるエイリアスを与えている．それ以降の行では，与えたエイリアスを通して，JS オブジェクトにポインタタグとオブジェクトタグを設定している．10 行目を例に説明すると，`normal_string` というエイリアスを通して，string 型のポインタタグを 100 に，オブジェクトタグを 4 に設定するとともに，100 を表すマクロとして `T_STRING` を，4 を表すマクロとして `HTAG_STRING` を定義している．

データ型のカスタマイズの利用方法について，15 行目の `normal_array` に，`T_GENERIC(000)` の代わりに `T_ARRAY(001)` を与えた場合について考える（リスト 2.2）．通常，JSValue を与えられてそれが例えば array 型オブジェクトであるかどうかを判定するには，リスト 2.3 の擬似コードで示すような手順が必要になる．`is_array` は，JSValue を受け取り，それが array 型オブジェクトのアドレスかどうかを判定する関数である．まず，`obj` がオブジェクトのアドレスであるかを確認するため，2 行目で，ポインタタグがある下位 3 ビットを取り出して，object 型を表す `T_GENERIC(000)` と比較する．object 型である場合，3 行目に入り，`obj` からオブジェクトのヘッダを取得する．`obj` に入っているのはオブジェクトのデータ本体の先頭アドレスであり，ヘッダはその 1 ワード手前にあるため，3 行目のように，`obj` から 1 ワード手前のアドレスを求め，そのアドレスにある 1 ワードを取得している．4 行目で，取得したヘッダの下位 8 ビットにあるヘッダタグを，整数値として取得している．最後に，5 行目でそのヘッダタグが，array 型を表す `HTAG_ARRAY(7)` であるかどうかを判定し，その結果を返す．このように，種類を判別するだけでもこれだけの段階を必要とする．しかし，`normal_array` にポインタタグとし

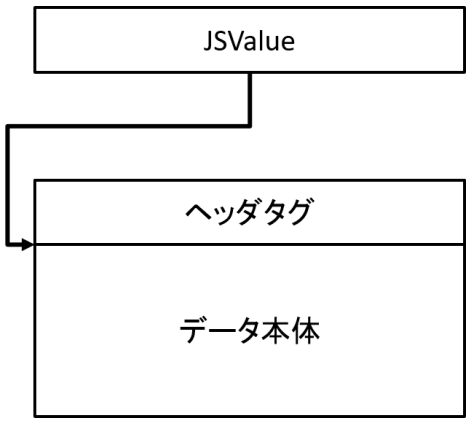

図 2.5　オブジェクトの構造

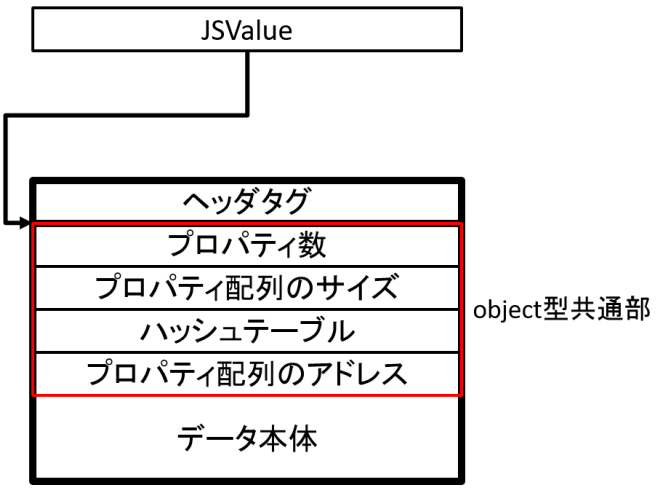

図 2.6　object 型オブジェクトの構造

```
1  string:                T_STRING(100)/HTAG_STRING(4)
2  flonum:                T_FLONUM(101)/HTAG_FLONUM(5)
3  special:               T_SPECIAL(110)
4  fixnum:                T_FIXNUM(111)
5  simple_object:         T_GENERIC(000)/HTAG_SIMPLE_OBJECT(6)
6  array:                 T_GENERIC(000)/HTAG_ARRAY(7)
7  function:              T_GENERIC(000)/HTAG_FUNCTION(8)
8  builtin:               T_GENERIC(000)/HTAG_BUILTIN(9)
9  regexp:                T_GENERIC(000)/HTAG_REGEXP(11)
10 normal_string:         +normal_string
11 normal_flonum:         +normal_flonum
12 normal_special:        +normal_special
13 normal_fixnum:         +normal_fixnum
14 normal_simple_object:  +normal_simple_object
15 normal_array:          +normal_array
16 normal_function:       +normal_funcition
17 normal_builtin:        +normal_builtin
18 normal_regexp:         +normal_regexp
```

リスト 2.1　データ型のカスタマイズ

て T_ARRAY(001) を定義していた場合，受け取った JSValue のポインタタグが T_ARRAY(001) だとわかった時点で，その JSValue が array 型オブジェクトのポインタであるとわかる．表 2.3 に示した通り，ポインタタグの空きは 3 通りしかないため，全てに固有のポインタタグを設定することはできないが，動作させるプログラムで多用されるオブジェクトに対して固有のポインタタグを定義して eJSVM をカスタマイズすることで，処理の高速化が図れる．

```
1  string:                +normal_string
2  flonum:                +normal_flonum
3  special:               +normal_special
4  fixnum:                +normal_fixnum
5  simple_object:         +normal_simple_object
6  array:                 +normal_array
7  function:              +normal_funcition
8  builtin:               +normal_builtin
9  regexp:                +normal_regexp
10 normal_string:         T_STRING(100)/HTAG_STRING(4)
11 normal_flonum:         T_FLONUM(101)/HTAG_FLONUM(5)
12 normal_special:        T_SPECIAL(110)
13 normal_fixnum:         T_FIXNUM(111)
14 normal_simple_object:  T_GENERIC(000)/HTAG_SIMPLE_OBJECT(6)
15 normal_array:          T_ARRAY(001)/HTAG_ARRAY(7)
16 normal_function:       T_GENERIC(000)/HTAG_FUNCTION(8)
17 normal_builtin:        T_GENERIC(000)/HTAG_BUILTIN(9)
18 normal_regexp:         T_GENERIC(000)/HTAG_REGEXP(11)
```

リスト 2.2　array 型のカスタマイズ

```
1  bool is_array(JSValue obj) {
2    if ((obj & 0x7) == 0) {              // 下位 3ビットが 000の場合
3      uintptr_t header = *((uintptr_t *)obj - 1);
4      size_t header_tag = header & 0xff; // ヘッダの下位 8ビットを取得
5      return header_tag == 7;
6    }
7    return false;
8  }
```

リスト 2.3　オブジェクトの種類の判別

また，1–9 行目について，エイリアスを与えない場合，その JS オブジェクトの定義は eJSVM から除かれる．例えば整数しか使わないプログラムでは flonum 型を定義せず，正規表現を使用しないプログラムでは regexp 型を定義しないことで，これらは仮想機械から取り除かれ，生成される eJSVM のプログラムサイズを小さくすることができる．

3 GC

GC とは，プログラム実行中，ヒープ中に動的に確保された領域のうち，不要になった領域を自動的に回収して再利用を可能とする機能のことである．その手法として特徴の異なる様々なアルゴリズムがある．2.4 節において eJSTK がデータ型についてカスタマイズできることを述べたが，本研究ではそれ以外に，メモリ管理の部分におけるカスタマイズを行えるようにした．具体的には，eJSTK に元々実装されているマークスイープ方式 GC に加えてコピー方式 GC を実装し，eJSVM が使用する GC アルゴリズムをマークスイープ方式とコピー方式の 2 種類から選択できるようにした．このとき，選択しなかった方のアルゴリズムのプログラムは eJSVM から除外される．

以下の節ではマークスイープ方式とコピー方式のアルゴリズムについて説明する．

3.1 マークスイープ方式

マークスイープ方式 GC は，プログラムが使用しているオブジェクト（以下，生きているオブジェクト）に印を付けるマークフェーズ（Mark phase），印の付いていないオブジェクトを不要なオブジェクト（以下，ごみ）として，その領域を回収するスイープフェーズ（Sweep phase）の 2 つのフェーズで構成される [8].

オブジェクトについて，プログラムがまだ使用しているかどうかは，ルート集合から他のオブジェクトへの参照を再帰的に辿っていったときに，そのオブジェクトに辿り着けるかどうかで判断する．ルート集合とは，プログラムが直接操作可能な領域のことで，スタックやレジスタ，ローカル変数，グローバル変数などが該当する．これらの領域から辿り着けないということは，プログラムがそのオブジェクトを参照できず，使用できないということを意味するため，そのオブジェクトはごみであると判断してよい．

これを踏まえ，マークフェーズではルート集合からオブジェクトへの参照を再帰的に辿りつつ，辿り着いたオブジェクトに印を付けていく．このとき，参照関係によっては同じオブジェクトに複数回辿り着くこともあるが，同じオブジェクトの子を何度も辿る必要はないため，辿り着いたオブジェクトに印が付いているかどうかを確認し，印が付いていた場合はそのオブジェクトの子を辿らない．

スイープフェーズではヒープを走査し，見つけたオブジェクトに印が付いていなかった場合，ごみとしてその領域を回収する．

このアルゴリズムには以下の特徴がある．

1. アルゴリズムが単純である．

2. 繰り返していくうちにヒープ内に不連続の小さな空き領域が発生する．
3. GC の時間がヒープサイズに依存する．

2 つ目の特徴について，そのようになった状態のことを断片化（fragmentation）と呼ぶ．オブジェクトは連続した空き領域にしか割り当てられないため，断片化が進行すると，空き領域の総量が十分にある場合でも，オブジェクトをヒープに割り当てることができなくなる．よって，GC としてマークスイープ方式のアルゴリズムのみを使用するプログラムは，長時間動作し続けるとメモリの使用効率が悪くなる．

3 つ目について，スイープフェーズではヒープ全体を走査するため，ヒープサイズに比例してスイープフェーズにかかる時間が増大し，結果として GC にかかる時間も大きくなる．

これらの特徴から，ヒープサイズが小さく，長時間動作させることのないような簡単なプログラムに向いている．

3.2 eJSVM のマークスイープ方式 GC

以下では，eJSVM に実装されているマークスイープ方式 GC について，擬似コードと共にアルゴリズムの詳細を述べる．

3.2.1 チャンクリスト

マークスイープ方式 GC は繰り返していくうちに，空き領域がヒープ上に散在するようになる．そうなったとき，ヒープ上の空き領域の位置と大きさを把握する必要がある．

eJSVM のマークスイープ方式 GC は，ヒープ上の空き領域を管理するのに，チャンクリストというデータ構造を用いている．チャンクは eJS オブジェクトと同じヘッダ構造を持ち，フィールドには次のチャンクへのポインタを持つ（リスト 3.1）．

アロケータは，ヒープ上に存在する，空き領域のそれぞれの先頭にチャンクをひとつずつ割り当て，そのチャンクが持つメンバ `next` を用いて，それらをひとつのリストとしてまとめて持つ（図 3.1）．先頭チャンクのアドレスはグローバル変数に格納される．各チャンクは，割り当てられた空き領域のサイズを `header` に持つ．オブジェクトを割り当てる際にはこのリストを辿り，割り当てに必要なサイズ以上の空き領域をファーストフィット方式で探し出す．リスト 3.2 に，

```
1 struct chunk {
2   header_t      header
3   struct chunk *next
4 }
```

リスト 3.1　チャンク

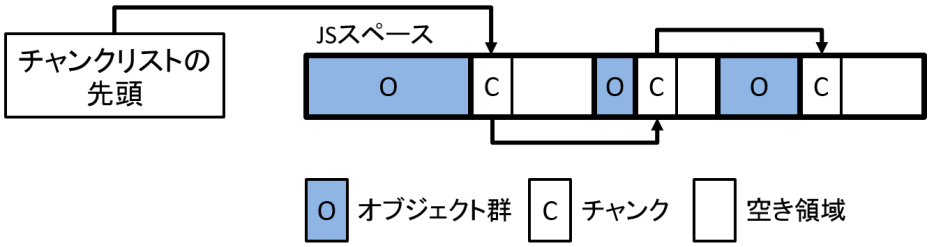

図 3.1　チャンクリスト

新しいオブジェクトのための領域を確保する擬似コードを示す．

3 行目でチャンクリストから先頭のチャンクを取り出し，4–30 行目を，十分な空き領域のチャンクを見つけるか，チャンクリストを辿り切るまで繰り返す．7 行目で `free_chunk` が管理している空き領域のサイズが，オブジェクトの割り当てに必要なサイズ以上あるかどうかを判定している．割り当てられるだけの空き領域があったならば，8 行目に入り，オブジェクトの割り当てに必要な領域を確保するための処理に進む．オブジェクトを割り当てる領域は，空き領域の末尾の方から確保する．このとき，オブジェクトを割り当てた後に，その後別のオブジェクトを割り当てられるだけの空き領域が残るかどうかで，処理を変える必要がある．8–16 行目はオブジェクトを割り当てた後に十分な空き領域が残らない場合の処理であり，17–26 行目は十分な空き領域が残る場合の処理である．前者の場合，その空き領域全体をオブジェクトの割り当てに使用する．12 行目で，空き領域の先頭にオブジェクトヘッダを割り当てている．このとき，ヘッダにはオブジェクトのサイズ，余った領域のサイズ，オブジェクトタグが格納される．そして 14 行目でその領域を管理していたチャンクをチャンクリストから外し，15 行目で割り当てたオブジェクトのデータ部分の先頭アドレスを返す．後者の場合，オブジェクトを空き領域の末尾に割り当て，チャンクの情報を更新する．20 行目でオブジェクトを割り当てた後の空き領域のサイズを計算し，21 行目でチャンクヘッダに記録されている空き領域の情報を更新する．22 行目でオブジェクトを空き領域の末尾に割り当てる際の先頭アドレスを計算し，24 行目でそのアドレスにオブジェクトヘッダをセットする．25 行目でオブジェクトのデータ本体の先頭アドレスを返して処理が終了する．

3.2.2　マークフェーズ

最初に行うルート集合のスキャンの擬似コードをリスト 3.3 に示す．

`root_scan` 関数はルート集合に含まれる全てのオブジェクトへの参照について `trace` 関数を呼ぶ．理由は 4.3 節で述べるが，eJSVM において，ルート集合にはオブジェクトへのアドレス以外に NULL も含まれるため，8 行目でそれをチェックしている．また，オブジェクトへのポインタを持っているデータ構造体は，オブジェクト以外へのポインタも内部に保持する可能性があ

```
1  void * allocate(size_t request_size, size_t type) {
2    struct space *prev = &chunk_list;
3    struct space *free_chunk = chunk_list.next;
4    while (free_chunk != NULL) { // チャンクリストを辿る
5      size_t free_size = get_size(free_chunk->header);
6      struct space *next = free_chunk->next;
7      if (free_size > request_size) { // 十分な空き領域がある場合
8        if (free_size < request_size + MINIMUM_CHUNK_SIZE) {
9          /* 要求サイズを切り取ると十分な空き領域が残らない場合 */
10         /* チャンク全体を割り当てに使用する */
11         /* オブジェクトヘッダをセットする */
12         set_header(free_chunk, request_size, free_size - request_size, type);
13         /* チャンクリストから外す */
14         prev->next = free_chunk->next;
15         return free_chunk + HEADER_SIZE;
16       }
17       else {
18         /* 要求サイズを切り出しても十分な空き領域が残る場合 */
19         /* 空き領域の末尾から必要サイズを切り出す */
20         size_t new_free_size = free_size - request_size;
21         set_size(free_chunk, free_size - request_size); // チャンクを更新する
22         uintptr_t object_addr = free_chunk + new_free_size;
23         /* オブジェクトヘッダをセットする */
24         set_header(object_addr, request_size, 0, type);
25         return object_addr + HEADER_SIZE;
26       }
27     }
28     prev = free_chunk;
29     free_chunk = free_chunk->next;
30   }
31   return NULL;
32 }
```

リスト 3.2　オブジェクトの割り当て

るため，9 行目で ptr の指す先が JS スペース内であるかどうか，すなわち，オブジェクトのアドレスであるかどうかを判定している．オブジェクトである場合には，それがスイープフェーズでごみとして回収されないよう，印を付ける必要がある．2.3.2 節で説明した通り，オブジェクトのヘッダにはサイズや型情報のほか，GC に必要な情報のための領域もある．マークスイープ方式ではその中の 1 ビットを印の有無の記録に使用し，このビットを“マークビット”と呼ぶ．10 行目でオブジェクトのヘッダのアドレスを取得し，11 行目で is_marked 関数にヘッダを渡している．is_marked 関数はヘッダにあるマークビットが立っているかどうかを判定する関数

```
1  void root_scan() {
2    foreach (obj in root_objs) { // ルート集合に含まれる全てのポインタ
3      trace(obj);                // 参照を辿る
4    }
5  }
6
7  void trace(void* ptr) {
8    if (ptr == NULL) return; // 何も参照していない場合は直ちに関数を抜ける
9    if (in_js_space(ptr)) {  // オブジェクトの参照であることを確認する
10     uintptr_t hdrp = (uintptr_t *)ptr - 1;
11     if (is_marked(*hdrp)) {
12       return;    // 参照先のオブジェクトに印が付けられていた場合は直ちに関数を抜ける
13     }
14     mark(hdrp); // 参照先のオブジェクトに印を付ける
15   }
16   /* 参照先のオブジェクトの子を辿る */
17   foreach (child in children(ptr)) {
18     trace(child);
19   }
20 }
```

リスト 3.3　ルートスキャン

である．ptr が指すオブジェクトについて，マークビットが 1 であれば，そのオブジェクトは既に辿られているため，12 行目でそのまま trace 関数を抜ける．マークビットが 0 の場合，14 行目で mark 関数にヘッダのアドレスを渡す．mark 関数は受け取ったアドレスからヘッダにアクセスし，マークビットを 1 にする．ptr がオブジェクト以外のデータ構造，または印のついていないオブジェクトを指していた場合には 16 行目に到達し，17–19 行目で，その構造体またはオブジェクトが持つ，全てのポインタに対して再帰的に trace 関数を呼び出し，オブジェクトを辿っていく．マークフェーズが終了すると，ヒープ中の生きているオブジェクトのヘッダには印が付いている．印の付いていないオブジェクトは生きていない（死んでいる）ため，次のスイープフェーズでその領域を回収する．

3.2.3　スイープフェーズ

スイープフェーズでは，ヒープを走査し，マークフェーズで印が付けられなかったオブジェクトに割り当てられている領域を回収する．リスト 3.4 はスイープフェーズの擬似コードである．

2 行目で sweep_pos を宣言するとともに，eJSVM におけるヒープである JS スペースの先頭アドレスで初期化し，7–33 行目にかけて JS スペースの端からオブジェクトを順に走査する．走査について，オブジェクトのヘッダからオブジェクトのサイズを取得し，その値だけアドレスを

```
1  void sweep(){
2    uintptr_t *sweep_pos = js_space.start;
3    uintptr_t header = *sweep_pos;
4    void *obj = sweep_pos + 1; // データ本体の先頭アドレス
5    chunk_list.next = NULL;    // チャンクリストを初期化する
6    struct space* prev = &chunk_list;
7    while (sweep_pos < js_space.start + js_space.size) {
8      { /* 使用中のオブジェクトのエリア */
9        while (sweep_pos < js_space.start + js_space.size
10              && is_marked(header)) {   // 印が付いている場合
11         reset_markbit(header);         // 印を消す
12         sweep_pos += get_size(header); // 次のオブジェクトのアドレスを取得する
13         header = *sweep_pos;
14         obj = sweep_pos + 1;
15       }
16     }
17     { /* ごみと空き領域のエリア */
18       void* free_start = sweep_pos;
19       while (sweep_pos < js_space.start + js_space.size
20              && !is_marked(header)) {  // 印が付いていない場合
21         sweep_pos += get_size(header); // 次のオブジェクトのアドレスを取得する
22         header = *sweep_pos;
23         obj = sweep_pos + 1;
24       }
25       if (free_start != sweep_pos) {   // ごみと空き領域のエリアが存在する場合
26         struct space *new_free_chunk = free_start; // 先頭にチャンクを割り当てる
27         set_size(new_free_chunk, sweep_pos - free_start);
28         new_free_chunk->next = NULL;
29         prev->next = new_free_chunk;
30         prev = prev->next;
31       }
32     }
33   }
34 }
```

リスト 3.4　スイープフェーズ

進めれば次のオブジェクトに辿り着く．空き領域の先頭にもチャンク構造によるヘッダがあるため，同様にして空き領域の次にあるオブジェクトの先頭に進むことができる．

JS スペースは使用中のオブジェクトのエリアと，ごみと空き領域のエリアに分けて考えることができ，なおかつ同種の連続したエリアをひとつのエリアとして考えれば，この 2 種類のエリアは交互に存在することになる．そのため，7–33 行目では，8–16 行目における使用中オブジェ

クトのエリアの走査と，17–32 行目のごみと空き領域のエリアの走査を交互に繰り返している．

8–16 行目では使用中のオブジェクトのエリアを走査しつつ，`reset_markbit` 関数によって，辿り着いたオブジェクトのマークビットを 0 に戻している．

17–32 行目ではごみと空き領域のエリアを走査し，エリアの開始地点と終了地点を特定する．25 行目でごみと空き領域のエリアがあったかどうかを確認し，あった場合，ごみと空き領域のエリアはまとめてひとつの連続した空き領域であるとみなせるため，26 行目から 28 行目で新しいチャンクを設定し，それを 29 行目でリストに繋げている．このチャンクリストは 5 行目の時点で一度リセットされている．

3.3 コピー方式

コピー方式 GC [5] は，生きているオブジェクトだけを別の領域にコピーし，元の領域を全て回収するアルゴリズムである．コピー方式において，ヒープは半分に分けられ，一方を from スペース，もう一方を to スペースと呼称する．新しいオブジェクトは from スペースにのみ割り当て，to スペースは使用しない．from スペースの空き領域が少なくなったとき，コピー方式 GC を起動する．マークスイープ方式と同様に，ルート集合からオブジェクトの参照を辿り，from スペースの生きているオブジェクトを to スペースにコピーする．そして最後に to スペースだった領域を新たに from スペースとし，from スペースだった領域を新たに to スペースとする．結果として，from スペースには生きているオブジェクトだけが残る．

このアルゴリズムには以下の特徴がある．

1. 割り当てにヒープの半分しか使用できない．
2. 断片化が発生しない．
3. 生きているオブジェクトの個数に GC の時間が依存する．

2 つ目について，to スペースにオブジェクトをコピーする際に領域の先頭から詰めて配置していくため，マークスイープ方式と異なり，断片化が発生しない．このため，空き領域を管理するためにチャンクリストを用いることもなく，空き領域の先頭アドレスを格納するためのポインタ変数をひとつ用意しておけばよい．

3 つ目について，生きているオブジェクトをコピーするのにかかる時間の比率が大きいため，ヒープ上のオブジェクトに対して，生きているオブジェクトの割合が大きいと GC にかかる時間が大きくなる．反対に，生成されたオブジェクトの大半がすぐにごみとなるようなプログラムでは高速に GC を実行できる．

4　実装

この章では eJSTK に実装したコピー方式 GC の詳細を説明する．

4.1　eJSVM のコピー方式 GC

リスト 4.1 に，本研究で実装したコピー方式 GC の擬似コードを示す．

`root_scan` は，基本的にはマークスイープ方式のものと同様に，ルート集合から `trace` 関数によって参照を辿っていく関数である．ただし，いくつか異なる点もある．3 行目について，`free_start` は空き領域の先頭アドレスが格納されているグローバル変数であり，`to_space_start` は to スペースの先頭アドレスが格納されているグローバル変数である．GC の実行中，オブジェクトを to スペースにコピーするために，to スペースの先頭を空き領域の先頭としている．4–6 行目はルート集合からオブジェクトを辿る処理であるが，マークスイープ方式のものと異なり，ルート集合から取り出しているのはオブジェクトのアドレスではなく，オブジェクトを参照するポインタのアドレスである．この理由は後述する．4–6 行目で from スペースの生きているオブジェクトは全て to スペースにコピーされるため，7–9 行目で from スペースと to スペースの先頭を交換している．

`trace` 関数について説明する．前述した通り，コピー方式の `trace` 関数が受け取るのはポインタ変数のアドレスである．13 行目で，`ptrp` が指すポインタから，中身のアドレスを `ptr` に取り出している．それ以降の処理は，17–29 行目を除けばリスト 3.3 に示したマークスイープ方式のルートスキャンとほぼ同じである．

17 行目でオブジェクトに印が付けられているかどうかを判定する．コピー方式は，マークスイープ方式同様に，ヘッダにあるマークビットを用いて，そのオブジェクトが既に辿られたかどうかを判別する．まだ印が付けられていない場合は，18–24 行目を通る．最初の 18 行目でマークビットを 1 にする．19 行目でヘッダを含めたオブジェクト全体のサイズを取得し，20 行目の `copy` 関数で from スペースにあるそのオブジェクトを to スペースの空き領域の先頭にコピーし，その分だけ，22 行目で空き領域の先頭を更新している．21 行目で to スペースにコピーしたオブジェクトの，データ本体の先頭アドレスを取得している．コピー方式 GC は実行前と実行後でオブジェクトのアドレスが異なるため，オブジェクトを参照するポインタ変数について，移動先のアドレスを参照するように更新する必要がある．23 行目でこのポインタ変数はコピー先のアドレスに更新されるが，他にこのオブジェクトを参照するポインタ変数が `trace` 関数に渡されたとき，そのポインタ変数を更新するため，コピー元のオブジェクトにコピー先のアドレスを記録しておく必要がある．そこで，24 行目で，コピー元のオブジェクトにコピー先のアドレスを格納する．コピー元に格納されたこのアドレスのことを“フォワーディングポインタ”と呼ぶ．

```
1  void root_scan() {
2    void *tmp;
3    free_start = to_space_start;  // オブジェクトをコピーする先のアドレス
4    foreach (ptrp in root_objs) { // ルート集合に含まれる全てのポインタのアドレス
5      trace(ptrp);                // 参照を辿る
6    }
7    tmp = from_space_start;       // to スペースと from スペースを入れ替える
8    from_space_start = to_space_start;
9    to_space_start = tmp;
10 }
11
12 void trace(void **ptrp) {
13   void *ptr = *ptrp;
14   if (ptr == NULL) return;
15   if (in_js_space(ptr)) {
16     uintptr_t *hdrp = (uintptr_t *)ptr - 1;
17     if (!is_marked(*hdrp)) { // 印が付いていない場合
18       mark(hdrp);            // 印を付ける
19       size_t size = get_size(*hdrp); // オブジェクトのサイズを取得する
20       copy(free_start, hdrp, size);  // オブジェクトを to スペースにコピーする
21       uintptr_t fp = (uintptr_t *)free_start + 1; // フォワーディングポインタ
22       free_start += size;
23       *ptrp = fp;
24       set_fp(ptr, fp); // コピー元オブジェクトにフォワーディングポインタを格納する
25       ptr = fp;
26     }
27     else { // ptr が指すオブジェクトは既にコピーされている
28       *ptrp = get_fp(ptr); // ポインタが指す先をフォワーディングポインタで更新
29       return;
30     }
31   }
32   for (child in children(ptr)) {
33     trace(child);
34   }
35 }
```

リスト 4.1　コピー方式 GC

最後に，32–34 行目でコピー先のオブジェクトが持つポインタ変数が辿られるよう，25 行目で，ptr の値をフォワーディングポインタで更新する．

一方，既に印が付けられていた場合は，27–30 行目を通る．マークスイープ方式では，印が付いていた場合には何もしないで関数を抜けていたが，コピー方式では，前述した通りコピー元の

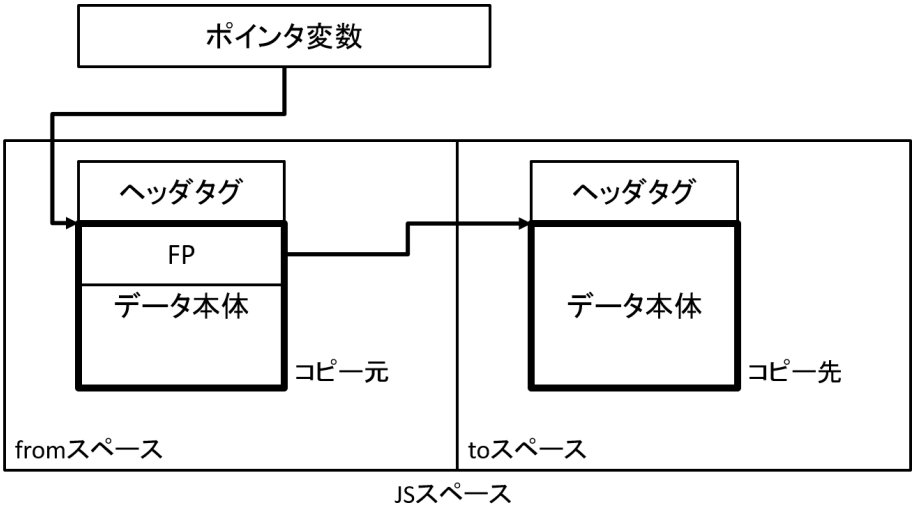

図 4.1　フォワーディングポインタの構造

```
1 void set_fp(void *ptr, uintptr_t fp) {
2   *ptr = fp;
3 }
4
5 void * get_fp(void *ptr) {
6   return *ptr;
7 }
```

リスト 4.2　フォワーディングポインタ

オブジェクトを指しているポインタは，コピー先を指すように更新する必要がある．29 行目で，ptr が指すオブジェクトからフォワーディングポインタを取得し，ptrp が指すポインタの値を更新している．

4.2　フォワーディングポインタ

ここでは，フォワーディングポインタについて説明する．本実装では，フォワーディングポインタはオブジェクトのデータ本体の先頭 1 ワードに格納する．図 4.1 にその様子を示し，リスト 4.2 に，フォワーディングポインタに関係する擬似コードを示す．元々持っていたデータは上書きされてしまうが，コピー元のオブジェクトのデータは既にコピー先にコピーした後であり，参照されることはないため，問題はない．ただし，一部のオブジェクトは，オブジェクトヘッダのみで構成され，データを持たない場合もあるため，eJSVM においてコピー方式 GC を使用する場合，新たにオブジェクトを割り当てる際，要求サイズにかかわらず最低でも 1 ワード分のデータ部を確保するようになっている．

4.3 ルート集合

ここでは，ルート集合について説明する．3.3 節で述べた通り，コピー方式もマークスイープ方式同様，生きているオブジェクトを特定するため，ルート集合に含まれるスタックやグローバル変数，ローカル変数などから参照を辿る．当然ながら，マークスイープ方式の処理に必要となるため eJSTK にも既に実装されていたが，コピー方式 GC を実装するにあたって，次に述べる理由からルート集合に手を加える必要があった．

4.1 節で説明した通り，コピー方式ではオブジェクトを参照していたポインタについて，そのオブジェクトをコピーした後に，コピー先のオブジェクトを参照するようにポインタの値を更新する必要がある．そのため，参照を辿る際にはオブジェクトのアドレスではなく，オブジェクトを参照するポインタ自体を受け渡す必要があった．従って，コピー方式のルート集合には，プログラムが直接参照できるポインタの，値ではなくそのポインタ自体のアドレスを含めるようにしなければならない．こうしたアルゴリズムの実装を見越して，eJSTK では元々ルート集合にはオブジェクトのアドレスではなく，オブジェクトを参照するポインタのアドレスを加えるように実装されていた．

一般に，あるオブジェクトを参照するポインタは複数存在することが考えられるが，マークスイープ方式であれば操作するのはオブジェクトのヘッダだけであるため，全ての生きているオブジェクトのアドレスさえ取得できれば，たとえルート集合に加えられていないポインタがあった場合でもプログラムは問題なく動作する．一方で，コピー方式が必要とするのは生きているオブジェクトを参照する全てのポインタである．そして，ルート集合の中でもローカル変数はプログラム中に遍在し，それらを余さずルート集合に含めなければ，オブジェクトがコピーされた後もコピー元のアドレスを参照するポインタが現れ，コピー方式は正しく動作しない．eJSTK にコピー方式 GC を実装した際，正しく動作しなかったため，ルート集合に加えられてない，すなわち，コピー先のアドレスに更新されないローカル変数が存在すると判断し，ルート集合の設定をやり直した．以下では，特にローカル変数の扱いについて述べる．

GC は，オブジェクトのためにヒープから領域を確保する際，ヒープの空き領域のサイズが一定以下のときに実行される．従って，オブジェクトを割り当てる処理が実行されるときには，その時点で参照できる全てのローカルポインタ変数がルート集合に含まれている必要がある．また，ルート集合に入れている間は，そのポインタが指すオブジェクトが残り続けるため，そのポインタ変数を使用しなくなった段階で，ルート集合から取り除く必要がある．リスト 4.3，4.4 にそれらの処理を行う擬似コードを示す．

リスト 4.3 ではルート集合として用いるためのスタックと，そのスタックへのプッシュとポップを行う関数を定義している．`gc_root_stack` はポインタ変数のアドレスを格納する配列で，`gc_root_stack_ptr` は `gc_root_stack` に格納されたデータの末尾の添字を示す．`gc_push`，`gc_pop` はそれぞれプッシュとポップを行う関数であるが，`gc_pop` について，11 行目でプログ

```
1  void **gc_root_stack[MAX_ROOTS];
2  int gc_root_stack_ptr = 0;
3
4  void gc_push(void **addr) {
5    gc_root_stack[gc_root_stack_ptr] = addr;
6    gc_root_stack_ptr += 1;
7  }
8
9  void gc_pop(void **addr)
10 {
11   if (gc_root_stack[gc_root_stack_ptr - 1] != addr) {
12     abort();
13   }
14   gc_root_stack_ptr -= 1;
15   gc_root_stack[gc_root_stack_ptr] = NULL;
16 }
```

リスト 4.3　ルート集合

ラムミスのチェックを行っている．先に述べた通り，ローカルポインタ変数はソースプログラム中に遍在し，プッシュとポップは必ず対応していなければならない．プッシュあるいはポップをし忘れるというプログラムミスを避けるため，gc_pop には最後にプッシュしたポインタ変数のアドレスを渡す．ポップしようとしているポインタ変数と，実際にスタックの一番上に格納されているポインタ変数が異なる場合は 12 行目に入り，プログラムエラーを知らせる．

リスト 4.4 に実際のプッシュとポップの利用例を示す．1–3 行目でマクロを定義している．GC_ROOT は，型名と変数名を指定することでその変数を宣言するマクロであり，その際に宣言した変数の値を 0 で初期化する．GC においてオブジェクトのポインタを辿る際，アドレスが初期化されていないポインタは予期しない動作の原因となってしまうため，ポインタ変数の宣言時に必ず初期化するようにしている．また，GC_PUSH，GC_POP は，ポインタ変数を指定して，そのアドレスをプッシュ，ポップするマクロである．アドレスを取得するための単項演算子である“&”を付け忘れ，ポインタ変数の値を渡してしまうことを防ぐために使用されている．

関数 f において，ルート集合に含むべきポインタ変数は v1，v2 である．7–8 行目で，これらの変数をルート集合のスタックにプッシュし，10 行目，12 行目で新たなオブジェクトを割り当てている．前述の通り，オブジェクトの割り当ての際には GC が発生する可能性がある．その際，7–8 行目でプッシュした v1，v2 がルート集合として参照される．v1，v2 は最後の 14–15 行目，すなわち v1，v2 がスコープから外れる直前に，スタックの一番上からそれらを順にポップしている．

この実装はプログラムミスを避けることを優先しているため，効率は良くない．例えば関数 f において，10 行目で GC が実行された際に v2 を辿ろうとするが，10 行目の時点で v2 にオブ

```
1  #define GC_ROOT(_type, _var) _type _var = ((_type) 0)
2  #define GC_PUSH(a) gc_push((void **)&a)
3  #define GC_POP(a) gc_pop((void **)&a)
4
5  void f(JSValue v1) {
6    GC_ROOT(JSValue, v2);
7    GC_PUSH(v1);
8    GC_PUSH(v2);
9    ...
10   v2 = allocate_new_object();
11   ...
12   v1 = allocate_new_object();
13   ...
14   GC_POP(v2);
15   GC_POP(v1);
16 }
```

リスト 4.4　ローカルポインタ変数のプッシュとポップ

ジェクトのアドレスは格納されていないため，その処理は無駄である．また，GC を呼ぶ可能性のある行が 10 行目と 12 行目であったとして，v2 を 13 行目以降で参照しない場合，12 行目の GC において v2 を辿る処理は不要である．そのため，効率を優先する場合には，GC の直前と直後に必要なポインタ変数だけを適宜プッシュ，ポップするべきであるが，本研究においては正しく動作することを目的とし，プログラムミスを避けることを優先した．

4.4　GC のカスタマイズ

本研究の eJSTK において，GC のアルゴリズムはプリプロセッサを通して制御される．具体的には，“MSGC”というマクロが定義されているときには，コピー方式のアルゴリズムはコードから取り除かれ，マークスイープ方式の GC を使用する eJSVM が生成される．反対に，“CGC”というマクロを定義した場合には，生成される eJSVM はコピー方式の GC を実行する．C 言語には CFLAGS というコンパイルオプションが存在し，定義したいマクロ名の先頭に“-D”を付けることでそのマクロを定義できる．従って，プログラマは eJSTK から eJSVM を生成する際に，CFLAGS に“-DMSGC”か“-DCGC”を指定することで，使用する GC アルゴリズムを選択できる．なお，どちらのマクロも定義されていない場合は，マークスイープ方式のアルゴリズムを使用する．

5　評価

eJSTK から，マークスイープ方式 GC を使用する eJSVM と，本研究で実装したコピー方式 GC を使用する eJSVM の 2 種類を生成し，表 5.1 に示す実行環境の下，それぞれで複数の同じベンチマークプログラムを動作させることで性能評価実験を行った．使用したベンチマークプログラムを表 5.2 に示す．以下の節では，実験によって測定した性能項目，実験の結果について述べる．

5.1　評価項目

eJSTK から，マークスイープ方式 GC を使用する eJSVM と，本研究で実装したコピー方式 GC を使用する eJSVM の 2 種類を生成し，それぞれで複数の同じベンチマークプログラムを動作させ，プログラムごとに以下の項目を計測した．

- プログラムの全実行時間
- GC 時間比率
- GC 停止時間
- GC の実行回数
- コードサイズ

プログラムの全実行時間（以下，全実行時間）は，プログラムの実行開始から実行終了までの長さを意味する．GC 時間比率は GC の処理時間，すなわち，プログラムが GC によって停止した時間の長さを表し，GC の処理速度の指標となる．全実行時間を $t1$，そのうちの GC を実行していた時間の合計を $t2$ としたとき，

$$\frac{t2}{t1}$$

で求められる．

GC 停止時間は GC1 回ごとの処理時間を意味し，リアルタイム性が求められるプログラムについて，レスポンスの速度の指標となる．

表 5.1　実行環境

OS	Ubuntu 17.10 (Linux kernel 4.13.0-31-generic)
CPU	Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
C コンパイラ	Clang 4.0.0
コンパイルオプション	`-O3`

表 5.2　ベンチマークプログラム

プログラム	説明
3d-cube	浮動小数点演算を中心とした三次元座標計算を行う.
3d-morph	浮動小数点演算を中心とした三次元座標計算を行う.
base64	Base64 エンコードを繰り返し，無作為に文字列を生成する.
binaryTree	二分木のボトムアップ手法による解析を行う.
bitops-bits-in-byte	単純なビット演算を繰り返す.
cordic	浮動小数点演算を中心とした数学的計算を行う.
fasta	遺伝子解析をモデルに，複雑な配列操作を繰り返す.
spectralnorm	浮動小数点演算を中心とした数学的計算を行う.
string-intensive	単純な文字列連結を繰り返す.
validateInput	文字列の生成，比較，連結等の操作を繰り返す.

さらに，メモリの使用効率の指標として，GC の実行回数も計測した．例えば，マークスイープ方式は繰り返すと断片化が発生してメモリの使用効率が悪くなるため，GC の回数が多くなりやすい．一方でコピー方式は，断片化こそ発生しないが，ヒープが半分に分けられ，そのうちの from スペースにのみオブジェクトを割り当てるため，プログラムが使用できるメモリはマークスイープ方式に比べて実質的に半分だけとなる．これらの要因等により，メモリを十全に使用できなくなれば，その分だけ GC が必要となる機会が増え，結果的に全実行時間及び GC 時間比率が増大する．

一般に，GC アルゴリズムの性能において重視される項目は，GC 時間比率と GC 停止時間である．GC 時間比率が小さければ，その分だけ全実行時間は短くなる．また，対話的なプログラムなど，リアルタイム性が求められる場合には，GC 停止時間は短いことが求められる．特に，重要な機器の制御プログラムにおいては，一定以上の停止時間が重大な事故に繋がりかねないため，GC 停止時間の最大値にも注目する必要がある．しかし，これらの性能は比例するものではなく，例えば GC 時間比率に優れた GC アルゴリズムが，GC 停止時間にも優れているとは限らない．加えて，GC アルゴリズムとプログラムには相性があるため，あるプログラムを実行したときに優れた GC 時間比率を出した GC アルゴリズムが，あらゆるプログラムで常に優れた GC 時間比率を出すとは限らない．本実験は，その確認を目的としている．

最後に，生成した 2 種類の eJSVM のコードサイズを比較した．本研究で実装した仕組みにより，使用しない方の GC アルゴリズムは eJSVM から取り除かれるため，選択したアルゴリズム次第で，生成される eJSVM のコードサイズがことなる．GC アルゴリズムの性能において，一般に GC 時間比率と GC 停止時間が重視されると述べたが，eJS が対象としている IoT デバイスは，利用できるメモリが限られている．従って，プログラムに適したアルゴリズムを選択する際には，生成される eJSVM のコードサイズの大小も重要となる．

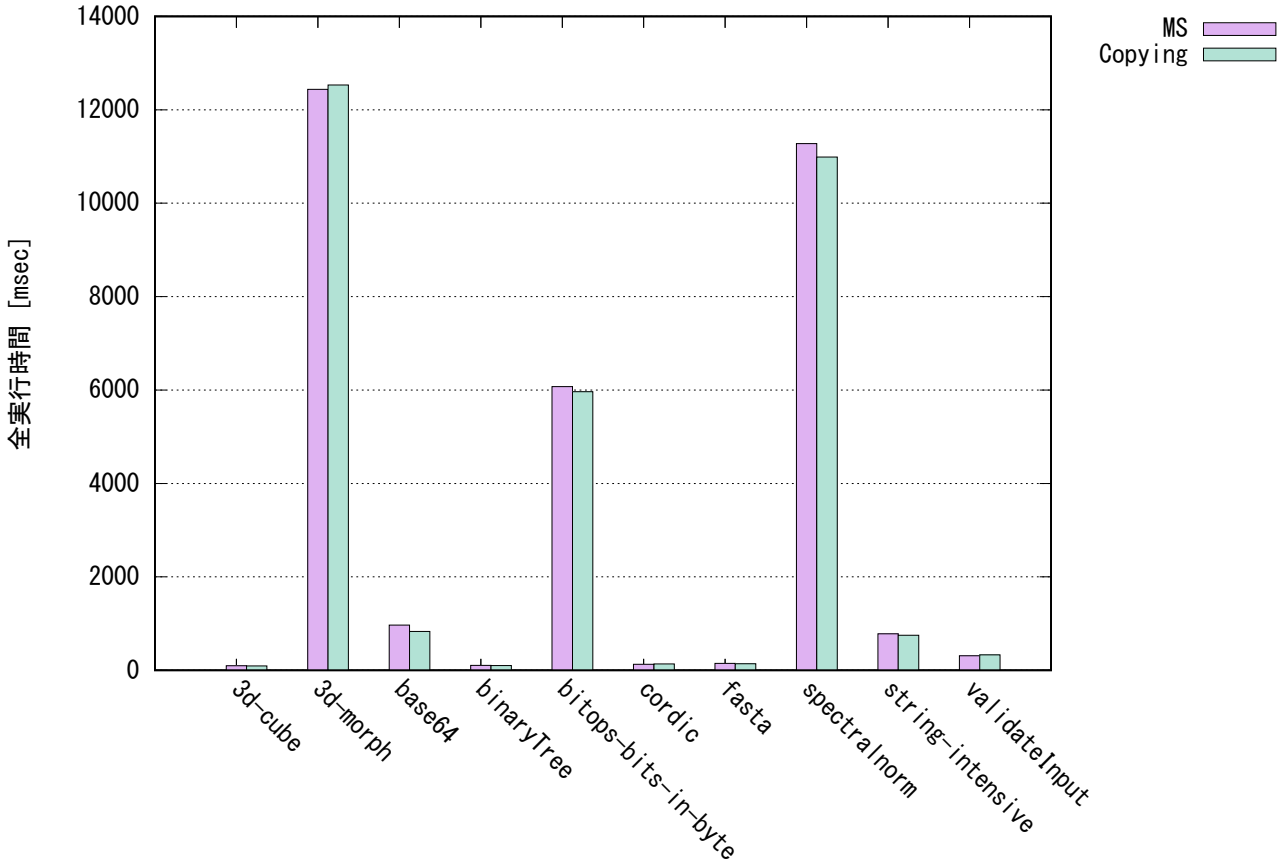

図 5.1　全実行時間の比較（JS スペース=10 メガバイト）

5.2　実験結果

実験は，ヒープである JS スペースのサイズを 10 メガバイトに設定した場合と，1 メガバイトに設定した場合の 2 パターンについて行った．初めに，JS スペースのサイズを 10 メガバイトに設定したときの結果について述べる．各プログラム，各 GC アルゴリズムについての測定結果を図 5.1，5.2，5.3，5.4 に示す．それぞれ，図 5.1 が全実行時間，図 5.2 が GC 時間比率，図 5.3 が GC 停止時間，図 5.4 が GC の実行回数を示す．いずれも，横軸は使用したプログラムの種類である．また，縦軸について，図 5.1 は単位がミリ秒であるが，図 5.3 と図 5.4 の単位はマイクロ秒である．なお，図においてマークスイープ方式，コピー方式はそれぞれ“MS”，“Copying”と略記している．

全実行時間と GC 時間比率について述べる．eJSVM にはキャッシュ等の特別なメモリアクセス手法は実装されていないため，基本的に全実行時間から GC の全停止時間を引いた，プログラム本体の処理時間は両 GC アルゴリズムでほぼ同じになる．従って，任意のプログラムについて，両 GC アルゴリズム間の，全実行時間の優劣と GC 時間比率の優劣は一致する．base64 と

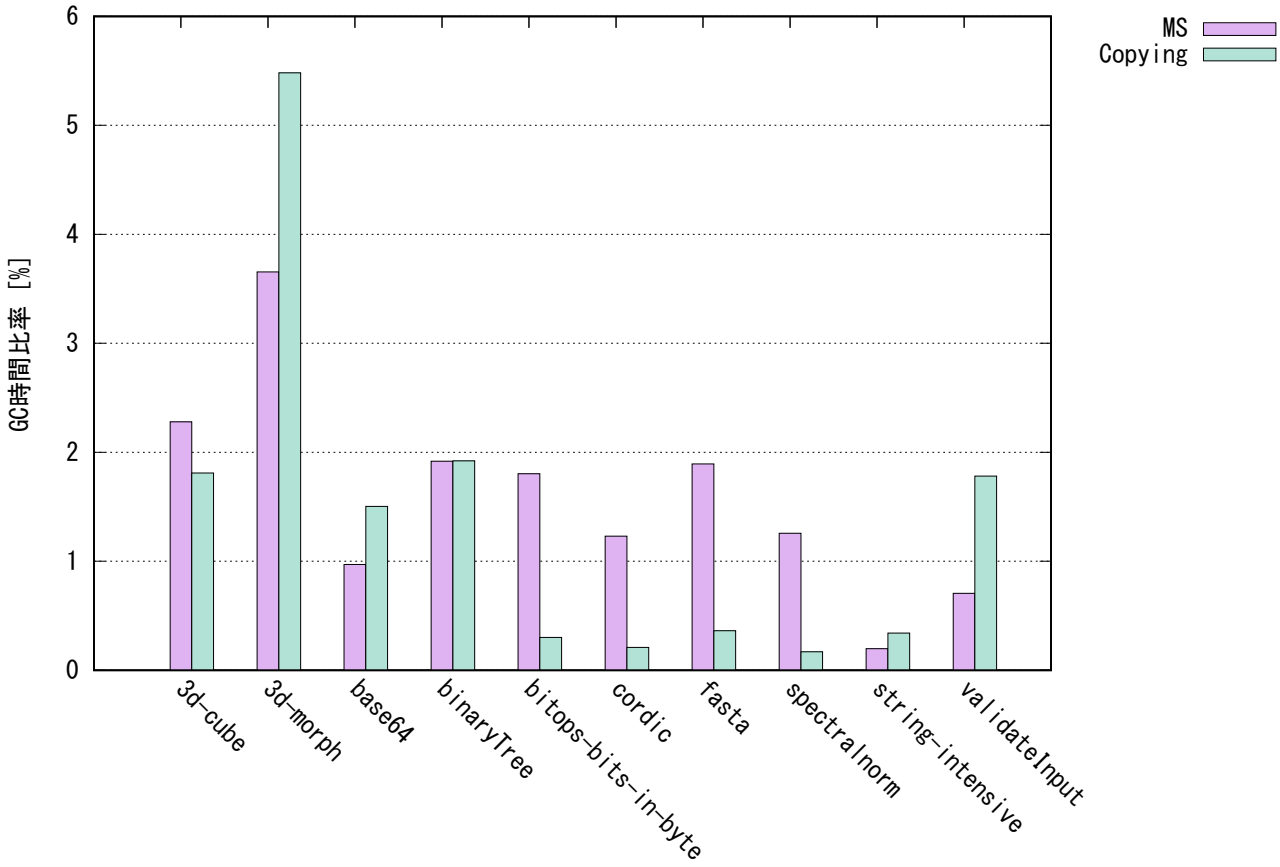

図 5.2　GC 時間比率の比較（JS スペース=10 メガバイト）

string-intensive において，全実行時間に優れた GC アルゴリズムと GC 時間比率に優れた GC アルゴリズムが一致していないのは，実行ごとに生じるプログラム本体の処理時間の誤差が原因であると考えられる．そのことを踏まえた上で計測結果を見ると，プログラムごとに，両 GC アルゴリズムの優劣が異なっていることが確認できる．これらは，プログラムと GC アルゴリズムの相性によるものだと考えられる．例えば，bitops-bits-in-byte，cordic，fasta，spectralnorm について，コピー方式の GC 時間比率はマークスイープ方式の 13%–19% である．これは，これらのプログラムが単純な計算を繰り返すもので，`Flonum` のような浮動小数点数オブジェクトが大量に生成され，それらがすぐにごみとして回収されたため，コピー方式においてコピーの回数が少なく済んだものと考えられる．一方で，3d-morph はコピー方式の GC 時間比率がマークスイープ方式の約 1.5 倍となっている．図 5.4 を見ると，GC の実行回数がマークスイープ方式の約 3 倍である．コピー方式は JS スペースを二分する上，3d-morph は三次元の物体の形を表すための座標を格納する巨大な配列を保持し続けるため，計算に使用できる JS スペースがさらに少ない．そのため，コピー方式の GC の実行回数が増え，そのことが GC 時間比率に影響したものと考えられる．

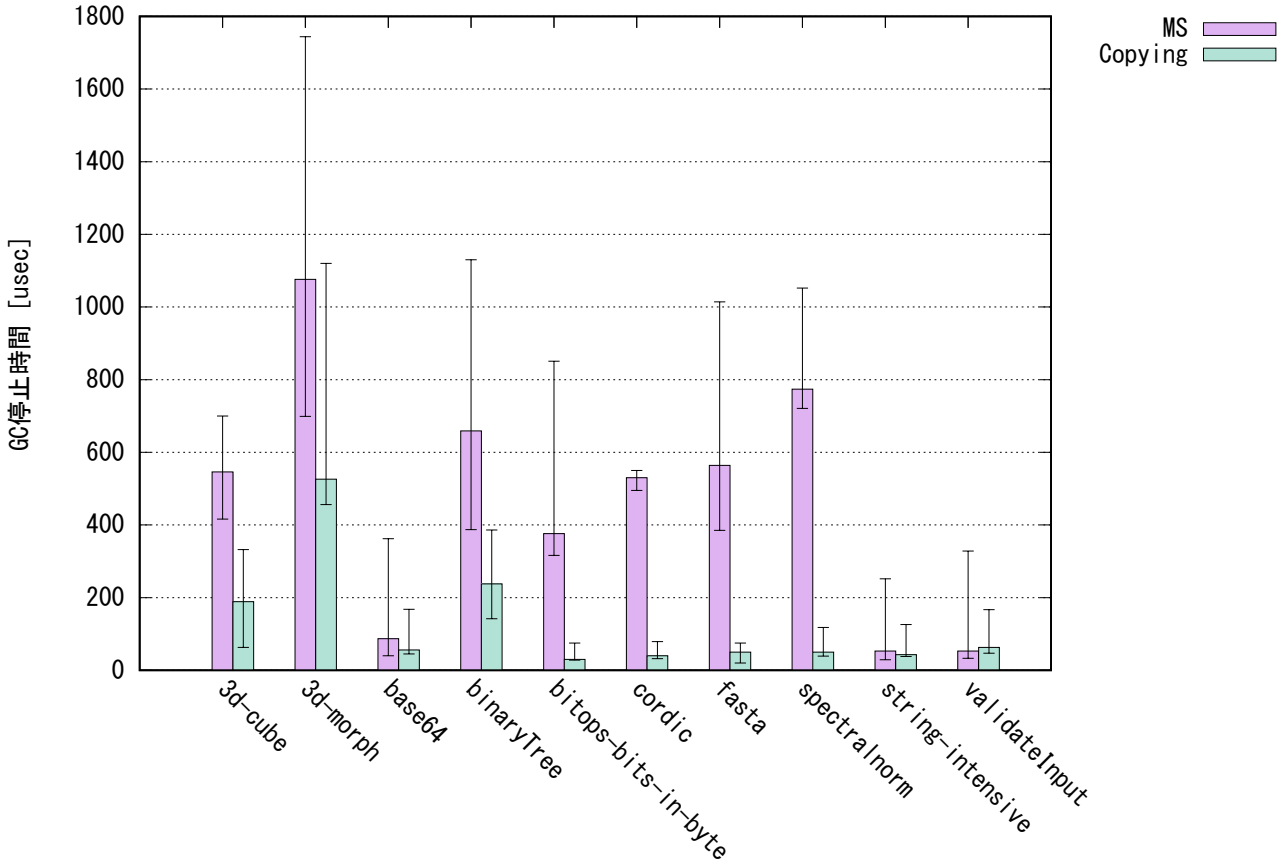

図 5.3　GC 停止時間の比較（JS スペース=10 メガバイト）

次に GC 停止時間に着目する．図 5.3 では，マークスイープ方式，コピー方式のそれぞれについて，棒グラフで GC 停止時間の平均値を，エラーバーで最大値と最小値を表している．全てのプログラムにおいてマークスイープ方式よりもコピー方式の方が，優れた GC 停止時間を示している．

最後に，GC の実行回数について述べる．マークスイープ方式は GC の回数が多くなると，断片化によってメモリの使用効率が悪くなり，さらなる GC の増加を誘発するはずだが，図 5.4 ではいずれもコピー方式 GC の回数が，マークスイープ方式の 2 倍以上となっている．先述した通り，2 倍という値は，コピー方式ではヒープである JS スペースを半分に分けていることが原因であると考えられ，さらにそのプログラムが最初から最後まで保持し続けるようなオブジェクトがあれば，実際に計算に使用できる領域は差がより大きくなる．マークスイープ方式で断片化の影響があまり見られないのは，今回実験に使用したプログラムは単純な処理を繰り返すものが多く，ヒープに割り当てられるオブジェクトのサイズが一定であったためであると考えられる．

次に，JS スペースのサイズを 1 メガバイトに設定したときの結果について述べる．各プログラム，各 GC アルゴリズムについての測定結果を図 5.5，5.6，5.7，5.8 に示す．なお，3d-cube，

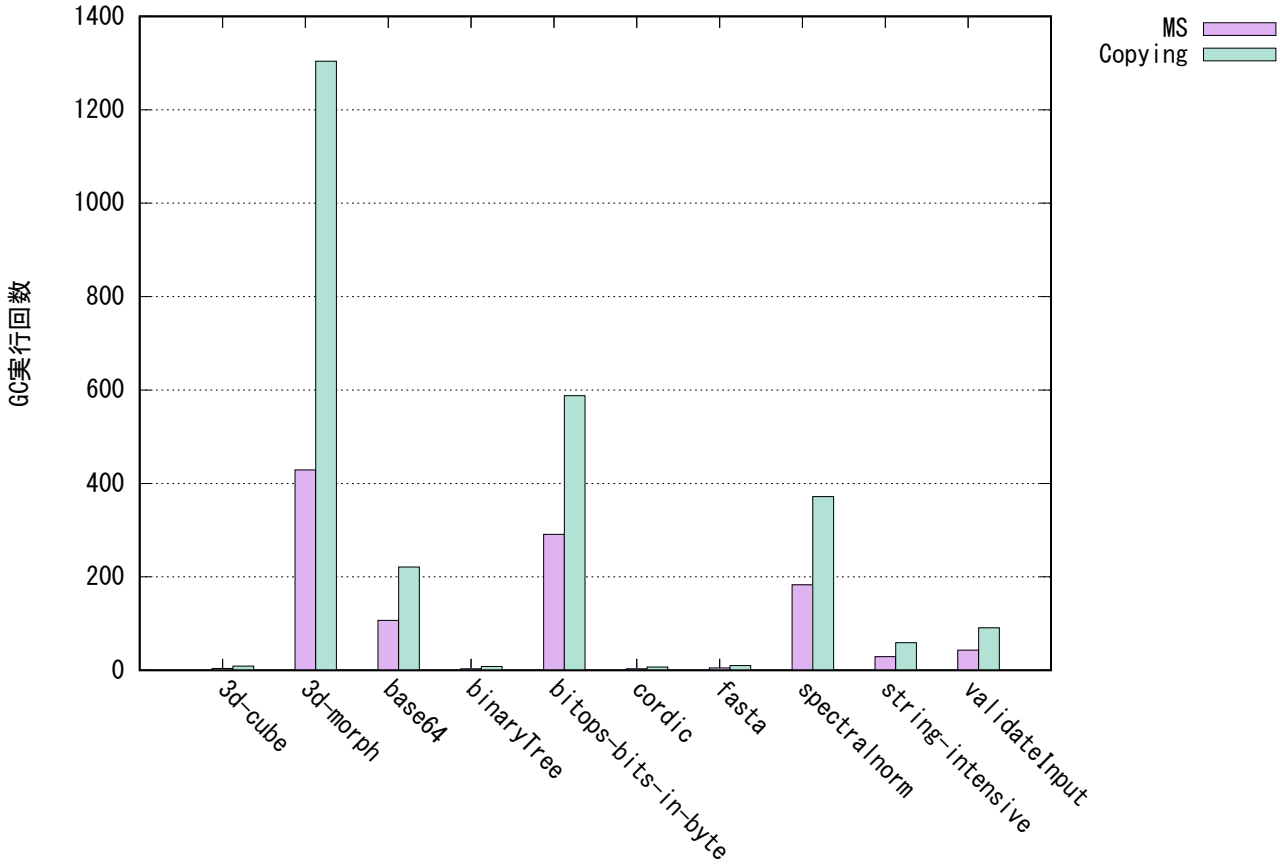

図 5.4　GC の実行回数の比較（JS スペース=10 メガバイト）

3d-morph，binaryTree は JS スペースが足りず，プログラムが途中でエラー終了したため，測定結果から除外している．

JS スペースが 10 メガバイトだったときの結果と GC 時間比率を比べると，コピー方式の方がどのプログラムにおいても増加が大きい．特に fasta はマークスイープ方式が約 1.4 倍になったのに対して，コピー方式は約 13 倍である．既に述べた通り，プログラムが実行開始から実行終了まで使用し続けるオブジェクトがあれば，その分だけプログラムが計算に利用できる領域は小さくなり，これは JS スペースが小さければ小さいほど影響が大きくなる．また，JS スペースが小さい分，GC を実行したときのオブジェクトの数も少ないため，GC 停止時間は軒並み小さくなっている．特に，GC の処理時間がヒープサイズに依存するマークスイープ方式は影響が大きく，base64，string-intensive，validateInput においてコピー方式よりも優れた結果を出している．

以上の実験結果から，プログラムごと，及び性能項目ごとに，マークスイープ方式とコピー方式のどちらが優れているかは異なることが確認できた．また，それらの結果は，さらに JS スペースのサイズによっても異なることが判明した．今回の実験では，JS スペースが 10 メガバイ

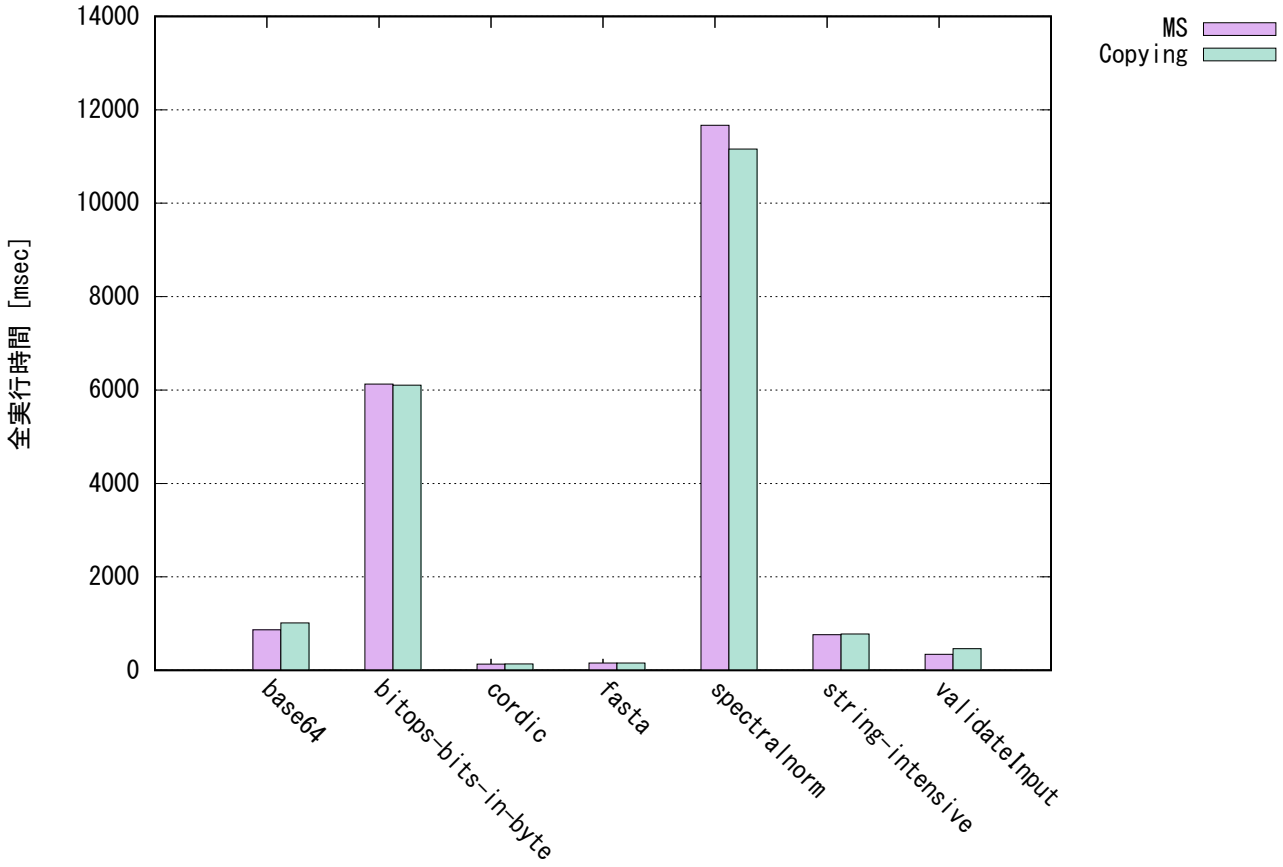

図 5.5　全実行時間の比較（JS スペース=1 メガバイト）

トのときにはコピー方式が優れた結果を出している場合が多かったが，1 メガバイトではその差が小さくなっていた．eJS が対象としている IoT デバイスでは利用できる資源が限られるため，JS スペースのサイズをさらに小さくする必要がある場合も十分に考えられ，そうした場合にはマークスイープ方式が有利になる場合も多くなることが予想される．特に，今回の実験において 1 メガバイトの設定で実行できずに結果から除外した 3d-cube，3d-morph，binaryTree のうち，3d-cube と binaryTree は，マークスイープ方式では実行ができていた．

最後に，生成された eJSVM のコードサイズを表 5.3 に示す．

表 5.3　eJSVM のコードサイズ

	マークスイープ方式	コピー方式
サイズ（キロバイト）	136	132

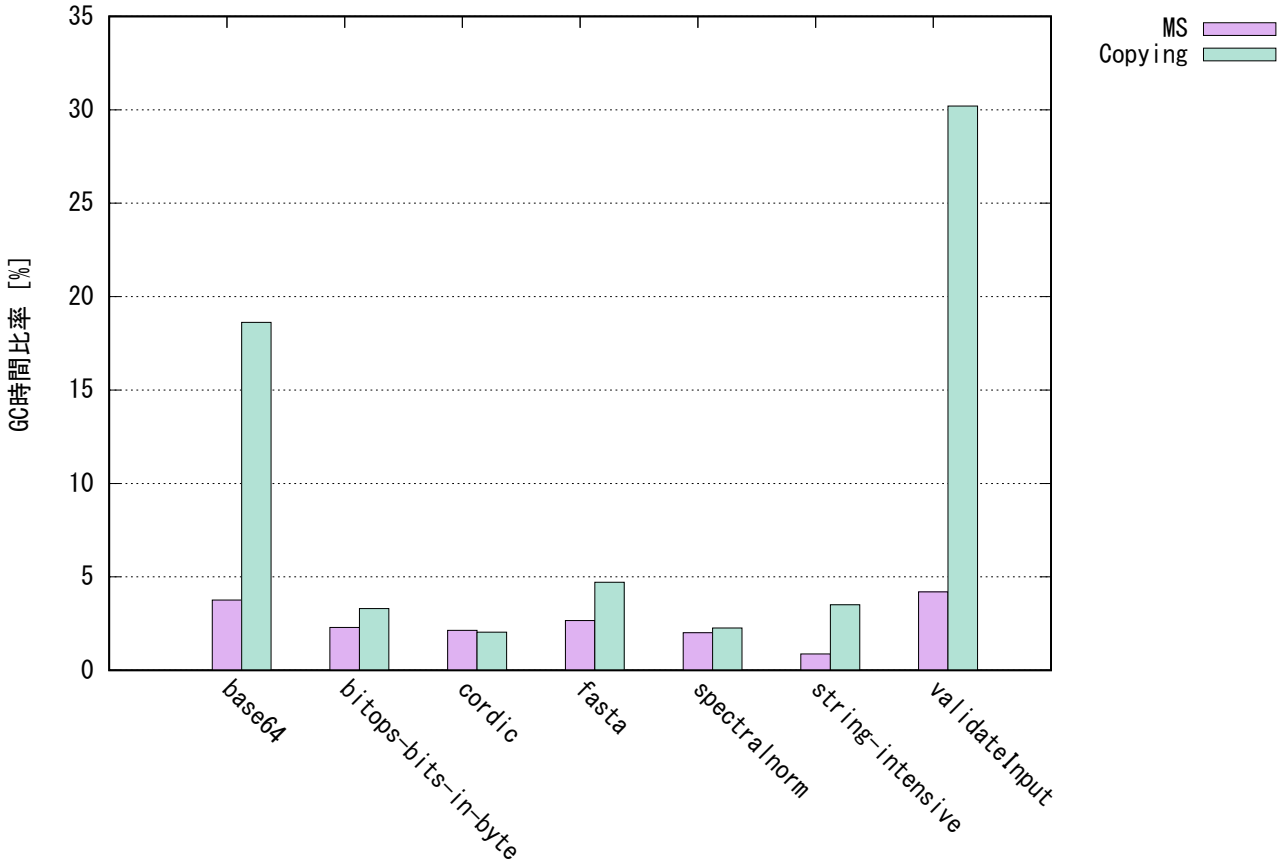

図 5.6　GC 時間比率の比較（JS スペース=1 メガバイト）

5.3　メモリオーバヘッド

コピー方式 GC において，フォワーディングポインタはオブジェクトのデータ本体の先頭 1 ワードに格納する．しかし，一部のオブジェクトは，オブジェクトヘッダのみを持ち，データ部分を持たないため，そのままではフォワーディングポインタを格納することができない．従って，本研究の実装では，要求サイズにかかわらず必ずデータ部分を 1 ワード以上確保することにしている．マークスイープ方式 GC と比べると，この部分がメモリオーバヘッドとなる．

実際に，各プログラムでこのメモリオーバヘッドがどれだけ発生したか，ヒープサイズを 10 メガバイトに設定して実験した．その結果を表 5.4 に示す．表において，“オブジェクト数”は，データ本体を持たないために 1 ワード余分に確保することになったオブジェクトの個数であり，“メモリオーバヘッド”は余分に確保した領域の総バイト数である．

プログラムによって差はあるが，多くのプログラムでメモリオーバヘッドは数十バイト程度に収まっている．また，3d-cube や cordic においては 100 キロバイトを超えるメモリオーバヘッドが発生しているものの，図 5.4 を見る限り，その影響は非常に小さい．実際，図 5.2 では，こ

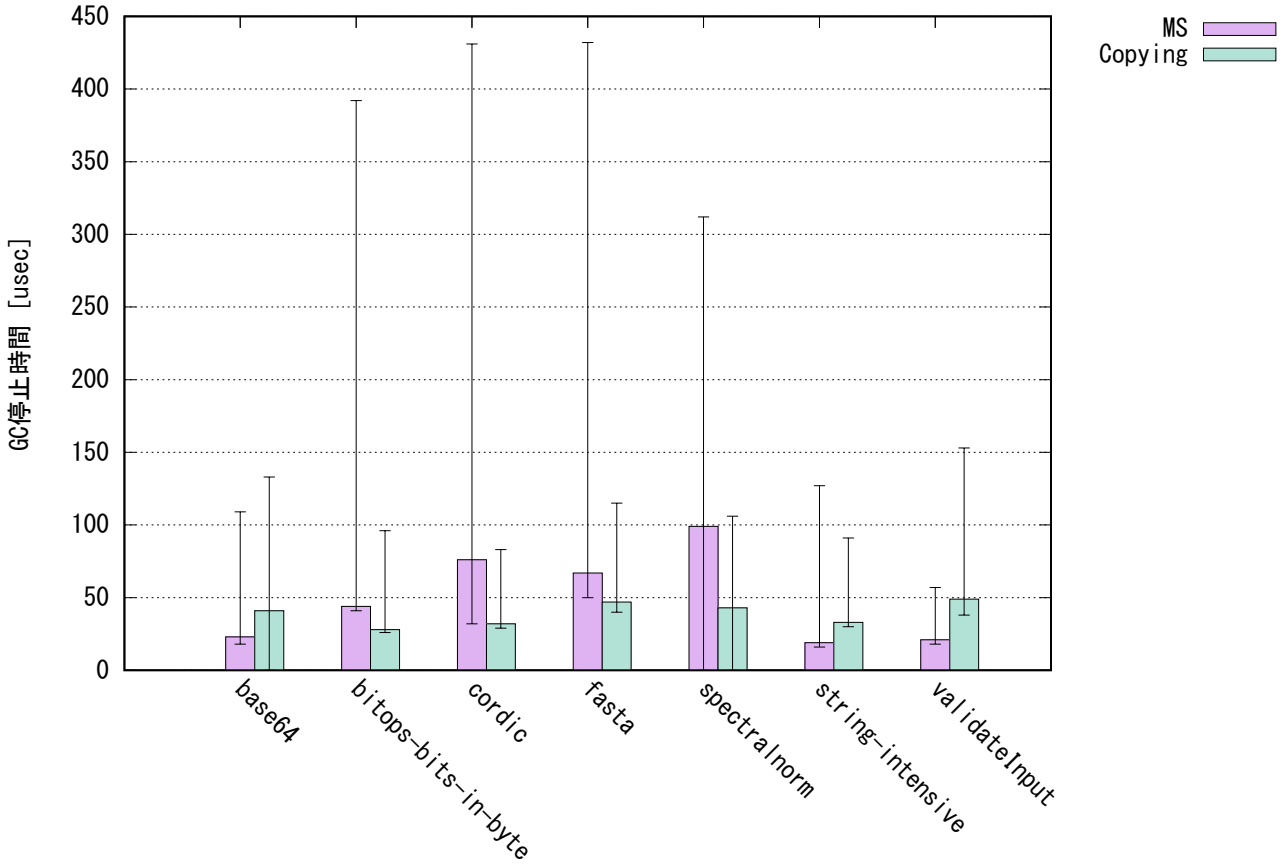

図 5.7　GC 停止時間の比較（JS スペース=1 メガバイト）

れらのプログラムにおいてコピー方式 GC の方が優れた結果を出している．

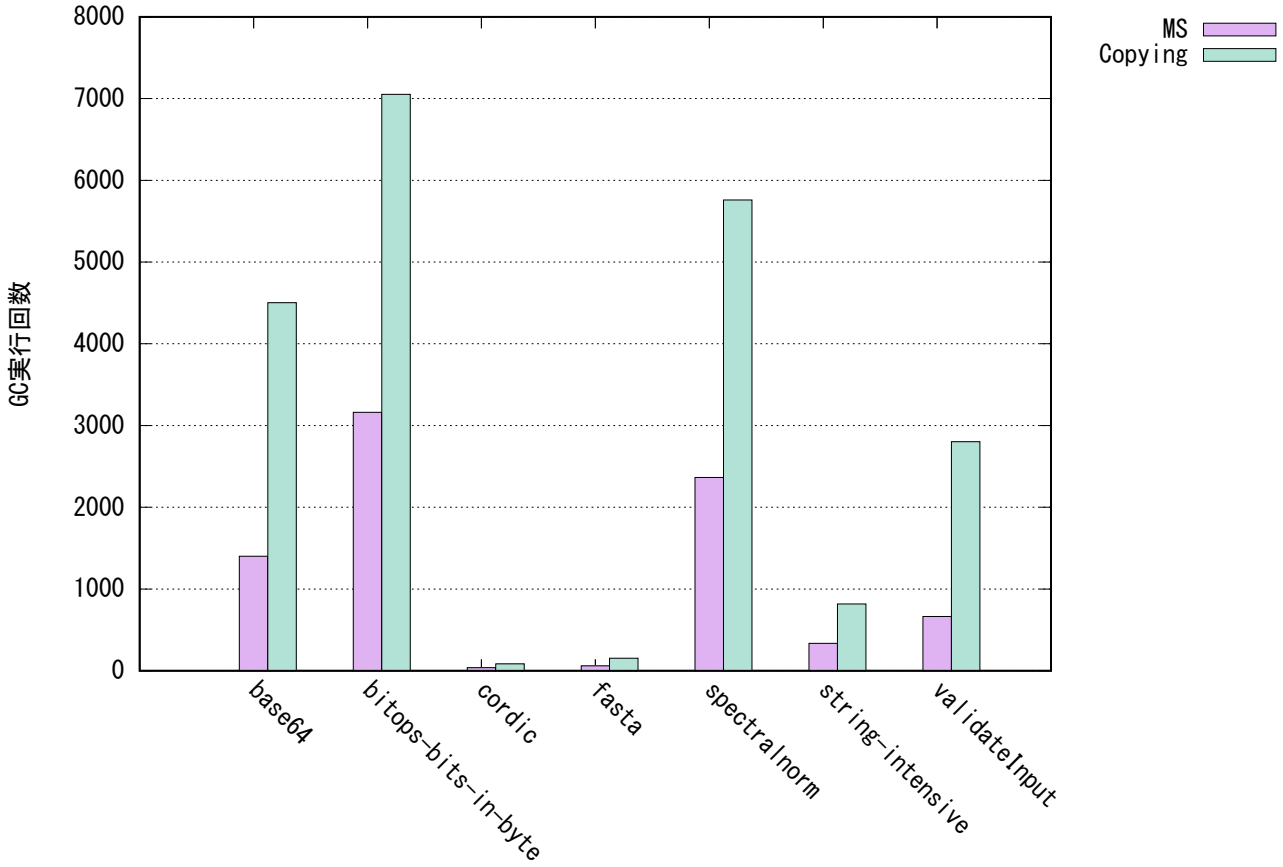

図 5.8　GC の実行回数の比較（JS スペース=1 メガバイト）

表 5.4　メモリオーバヘッド

	オブジェクト数	メモリオーバヘッド（バイト）
3d-cube	20,848	166,784
3d-morph	5	40
base64	5	40
binaryTree	4	32
bitops-bits-in-byte	4	32
cordic	25,005	200,040
fasta	7	56
spectralnorm	7	56
string-intensive	5	40
validateInput	8	64

6 関連研究

Vmgen [4, 7] は，主にスタックベースの仮想機械を生成するフレームワークである，命令仕様の定義を与えることで，その命令の処理に対応した仮想機械を生成できる．Vmgen と eJSTK はどちらもカスタマイズした仮想機械を生成するフレームワークであるが，カスタマイズの方向性が異なる．Vmgen の場合は，プログラマが命令仕様を定義することで，仮想機械で使用される命令セットをカスタマイズすることができる．eJSTK の場合は，データ型について定義することで，仮想機械で使用されるデータ型をカスタマイズすることができる．GC について，Vmgen には実装されていないため，使用する場合には実行時ライブラリとして用意する必要がある．一方で，eJSTK はマークスイープ方式とコピー方式の 2 種類のアルゴリズムが実装されており，それらが選択可能である．加えて，GC 中はオブジェクトへのアクセスが頻繁的に行われるため，2.4.1 節で述べたオブジェクトアクセスの高速化の恩恵が得られる．

mruby [1] は組込み機器に最適化された軽量なスクリプト言語である．こちらはインクリメンタル方式と世代別方式 GC が実装されている．本研究では元から実装されているマークスイープ方式に加えてコピー方式 GC を実装した．5.1 節で説明した通り，GC について，アルゴリズムごとの優劣は一概につけられないが，eJSTK に実装された 2 種類の方式は比較的シンプルなアルゴリズムであり，その分だけコードサイズは小さくなりやすいため，資源の限られた IoT デバイスを対象としている eJSTK には適している．また，eJSTK の GC はデータ型のカスタマイズと併用することで，前述した速度面での恩恵に限らず，コードサイズ面で最適化した GC コードを生成することができ，仮想機械のさらなる軽量化が図れる．

7　結論

IoT デバイスをターゲットとした eJSVM について，それを提供するフレームワークである eJSTK にコピー方式 GC アルゴリズムを実装し，既存のマークスイープ方式と選択可能にした．

マークスイープ方式 GC を使用する eJSVM と，コピー方式 GC を使用する eJSVM のそれぞれで，複数のベンチマークプログラムを動作させて性能を比較したところ，ヒープ領域である JS スペースのサイズ，動作させるプログラム，比較する性能項目ごとに，優れた GC アルゴリズムは異なるということが確認できた．従って，IoT デバイスの性能や動作させるプログラムに応じて，使用する GC アルゴリズムを選択できるようにすることは重要である．

参考文献

[1] T. Azumi, Y. Nagahara, H. Oyama, and N. Nishio. mruby on TECS: component-based framework for running script.

[2] G. Chadha, S. Mahlke, and S. Narayanasamy. Efetch: Optimizing instruction fetch for event-driven webapplications. In *Proc. 23rd International Conference on Parallel Architectures and Compilation*, PACT 2014, pages 75–86, 2014.

[3] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification.* 5.1 edition, June 2011.

[4] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. Vmgen - a generator of efficient virtual machine interpreters. *Softw., Pract. Exper.*, 32(3):265–294, 2002.

[5] R. R. Fenichel and J. C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *ACM Comm.*, v.12 n.11, 611–612, November 1969.

[6] 藤井 章博. IEEE 論文に基づく IoT 研究動向の計量書誌学的調査. `http://hdl.handle.net/11035/3092`, 科学技術動向;149, 2015.

[7] D. Gregg and M. A. Ertl. A language and tool for generating efficient virtual machine interpreters. In *Proc. Domain-Specific Program Generation*, pages 196–215, 2003.

[8] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *ACM Comm.*, v.3 n.4, pages 184–195, April 1969.

[9] T. Kataoka, T. Ugawa, and H. Iwasaki. A Framework for Constructing JavaScript Virtual Machines with Customized Datatype Representations. *SAC 2018*, April 9–13, 2018, Pau, France.

[10] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi. Microarchitectural implications of event-driven server-side web applications. In *Proc. 48th International Symposium on Microarchitecture*, MICRO-48, pages 762–774, 2015.

謝辞

本研究を行うにあたり，終始変わらぬ御指導を賜りました岩崎英哉教授に深く感謝致します．また，本研究に対して多大な御助言を頂きました，高知工科大学の鵜川始陽准教授，そして岩崎研究室，中野研究室の皆様に心から御礼申し上げます．