令和 4 年度

修士学位論文

組み込みシステム向け JavaScript 仮想機械の文字列オブジェクトの実装戦略

Implementation Strategies of String Objects in JavaScript Virtual Machines for Embedded Systems

1255112　近森 凪沙

指導教員　高田 喜朗

2023 年 2 月 28 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

要 旨

組み込みシステム向け JavaScript 仮想機械の文字列オブジェクトの実装戦略

近森 凪沙

組み込みシステム向け JavaScript 処理系 eJS では，全ての文字列を正規化しハッシュ表で管理している．これにより，同一の文字列はメモリを共有することができ，文字列の比較が高速化されるメリットがある．一方で，正規化のためのハッシュ表が必要となる．eJS では、ハッシュ表の他に JavaScript のオブジェクトを格納するヒープや，プロパティ検索を高速化するために使用される領域，JavaScript から呼び出されるネイティブ関数によって使われる領域などがランダムアクセスメモリ（RAM）に配置されている．組み込みシステムは RAM の容量が少ないため，なるべく RAM の消費を抑える必要がある．

JavaScript プログラムの実行前に生成される文字列は，フラッシュメモリに配置するため RAM を消費しないが，ハッシュ表はプログラム実行中に生成される文字列も管理するため，フラッシュメモリに配置することができない．

そこで，本研究ではハッシュ表を使用しない文字列管理方式を提案する．プログラム実行前に生成される文字列だけをコンパイル時に正規化し，実行中に生成される文字列は同一のものであっても個別にメモリを割り当てるようにする．文字列を比較する時は，正規化した文字列同士は文字列のアドレスによる比較を行い，個別にメモリを割り当てた文字列との比較は文字列の内容で比較を行うようにした．また，一般的な JavaScript プログラムの実行では，オブジェクトのプロパティ検索が最も頻繁に行われる文字列の比較を伴う操作である．それを高速化するために，コンパイル時の解析により文字列の比較方法を決定する方法と，インラインキャッシュに保存されたプロパティかどうか判定する際には文字列のアドレ

スによる比較のみを行う方法の，2 種類の高速化法を考案した．

提案方式の効果を確認するため，提案方式を実装しベンチマークプログラムに適用した．その結果，提案方式を用いることにより RAM の空き領域や JavaScript のオブジェクトを格納するヒープの空き領域が増加した．一方，実行時間は既存の eJS と比べて平均で 6％，最大で 86％増加した．また，コンパイル時の解析により文字列の比較方法を決定する方法を用いると，実行時間は既存の eJS に比べて平均で 2％減少し，キャッシュされたプロパティとの比較で文字列のアドレスによる比較のみを行う方法では，実行時間は既存の eJS と同程度の実行時間となった．なお，実行時間が 80％以上増加したプログラムは，プロパティ名に動的文字列を大量に使用する人工的なプログラムであり，このような特殊な場合には提案方式は効果がないことがわかった．

キーワード　eJS, Mbed, JavaScript, 文字列オブジェクト

Abstract

Implementation Strategies of String Objects in JavaScript Virtual Machines for Embedded Systems

Nagisa Chikamori

In eJS, a JavaScript processor for embedded systems, all strings are normalized and managed in a hash table. This allows identical strings to share memory and has the advantage of speeding up string comparisons. On the other hand, a hash table is required for normalization. In addition to the hash table, eJS has a heap that stores JavaScript objects, an area used to speed up property lookups, and an area used by native functions called from JavaScript in random access memory (RAM). Embedded systems need to consume as little RAM as possible because RAM capacity is small.

Strings generated before the execution of a JavaScript program do not consume RAM because they are placed in flash memory. However, a hash table cannot be placed in flash memory because it also manages strings generated during the execution of a JavaScript program.

Therefore, we propose a strings management scheme that does not use a hash table. Only strings generated before the execution of a JavaScript program are normalized at compile-time, and strings generated during the execution of a JavaScript prgoram are allocated memory individually, even if they are identical. In string comparison, normalized strings are compared by the address, while strings with individually allocated memory are compared by string content. In addition, the general execution of a JavaScript program, object property lookups are the most frequent operations involving

string comparisons. To speed up this process, we have devised two speed-up methods. The first method determines the string comparison method based on compile-time analysis. The second method only does a string comparison by address when determining if a property is stored in the inline cache.

We implemented the proposed method and applied it to benchmark programs to confirm its effectiveness of the proposed method. As a result, using the proposed method increased the free space of RAM and a heap that stores JavaScript objects. On the other hand, execution time increased by an average of 6 % and a maximum of 86 % compared to existing eJS. Moreover, using a method that determines the string comparison method based on compile-time analysis, the execution time was reduced by an average of 2 % compared to existing eJS. A method only does a string comparison by address when determining if a property is a cached one, the execution time was about the same as the existing eJS. The programs that increased the execution time by more than 80 % were artificial programs that used a large number of dynamic strings in the property names. In such special cases, the proposed method proved to be ineffective.

key words eJS, Mbed, JavaScript, string objects

目次

第 1 章　はじめに　1

第 2 章　組み込みシステム向け JavaScript 処理系 eJS　4

2.1　eJSVM 4

2.2　JavaScript ヒープ 6

2.3　インラインキャッシュ 6

2.4　文字列 7

2.4.1　文字列オブジェクト 7

2.4.2　静的文字列と動的文字列 7

2.4.3　eJSVM の文字列管理方式 9

2.4.4　文字列に対する処理 11

第 3 章　Mbed　13

3.1　Mbed の全体像 13

3.2　FRDM-K64F 13

3.3　メモリマップ 14

第 4 章　既存の文字列管理方式　16

4.1　個別生成方式 16

4.1.1　予備実験 16

4.1.2　文字列の比較回数の調査 18

第 5 章　新しい文字列管理方式の提案　20

5.1　ハイブリッド文字列管理方式 20

5.1.1　動的な文字列比較 21

5.2 動的な文字列比較の高速化 22
5.2.1 静的な文字列比較 23
5.2.2 楽観的な文字列比較 25

第 6 章 評価 28
6.1 ベンチマーク 28
6.2 RAM の使用状況 29
6.3 JS ヒープの空き領域 29
6.4 実行時間 31

第 7 章 関連研究 33

第 8 章 まとめ 35

謝辞 37

参考文献 38

図目次

2.1 eJSVM の生成過程 5
2.2 オペランド仕様 6
2.3 文字列オブジェクトのデータ構造 7
2.4 フラッシュメモリ中の文字列オブジェクト 9
2.5 静的文字列と動的文字列の例 9
2.6 eJSVM の文字列管理方式 10

3.1 FRDM-K64F 14
3.2 SRAM_L 15
3.3 SRAM_U 15

4.1 インターンする方式と個別生成方式の実行時間の比 17
4.2 1 秒当たりの文字列の比較回数 19
4.3 文字列の比較回数と実行時間の比の関係 19

5.1 ハイブリッド文字列管理方式 21
5.2 動的な文字列比較 22
5.3 static 系命令への置き換え 24
5.4 楽観的な文字列比 27

6.1 `.heap_0` セクションと `.heap` セクションに割り当てているデータの総計 . . 30
6.2 eJSVM 起動直後の JS ヒープの使用状況 31
6.3 実行時間の比（左からインターンする方式，ハイブリッド文字列管理方式，静的な文字列比較，楽観的な文字列比較） 32

表目次

3.1 FRDM-K64F の性能 14

5.1 各 static 系命令の引数 24

6.1 実験環境 28

第 1 章

はじめに

近年，IoT（Internet of Things）技術の普及により，組み込みシステムの利用が広まっている．組み込みシステムのアプリケーション開発では，C 言語やアセンブリ言語が用いられることが多い．しかし，これらの低抽象度の言語による開発は，コードが複雑化しやすいため誤りなく記述することが難しく，時間もかかる．そのため，開発者の負担となってしまう．高級言語である JavaScript は，C 言語やアセンブリ言語と比べると開発が容易であるため，JavaScript で組み込みシステムのアプリケーションを開発することができれば，開発者の負担を減らすことができると考える．

そこで、JavaScript で組み込みシステムのアプリケーションが開発できるようにするための研究が行われている．その一つに，我々が開発を進めている JavaScript 処理系 eJS[1] がある．eJS が提供する仮想機械（以下，VM）である eJSVM は，組み込みシステムで実行することを想定して生成される小型の JavaScript VM で，100 KB 程度のメモリで実行することができる．

組み込みシステムには，フラッシュメモリと RAM の両方を搭載しているものがある．そのようなシステムでは，RAM の容量が小さい一方で，フラッシュメモリは書き換え不可なものが多い．そのため，大きいプログラムを実行するためには，RAM を効率的に使用する必要がある．eJSVM では，JavaScript のオブジェクトを格納するためのヒープや，オブジェクトのプロパティへのアクセスを高速化するインラインキャッシュの領域，JavaScript から呼び出されるネイティブ関数によって使用される領域や，文字列を管理するためのハッシュ表などを RAM に配置している．

そこで，本研究では利用可能な RAM 領域を増やすため，文字列を管理するハッシュ表を

使用しない文字列の管理方式を提案する．JavaScriptでは，プログラム中にある関数名やプロパティ名，文字列リテラルなど文字列が多く使用されている．eJSVMでは，同じ文字列を一つにまとめて共有するインターンを行っており，文字列をハッシュ表で管理している．文字列が必要な時はハッシュ表を検索し，ハッシュ表に登録されている場合は登録されている文字列を使う．これにより同じ文字列に割り当てるメモリが削減され，文字列の比較はアドレスで比較するため高速化される．しかし，この方式はハッシュ表などがRAMを消費する．ハッシュ表を使用しない文字列管理方式もあるが，文字列をインターンせず個別にメモリを割り当てるようになるため，文字列のアドレスによる比較ができず遅くなる．

eJSでは，JavaScriptプログラム中に明示的に記述された文字列（文字列定数やプロパティ名など）は，コンパイル時にeJSVMに直接埋め込み，フラッシュメモリに配置する．フラッシュメモリに配置している文字列は既にインターンしているため，文字列のアドレスによる比較ができる．インターンしている文字列同士だけでもアドレスによる比較ができれば，比較を一部高速化することができると考える．

本研究ではプログラム実行前に生成される文字列のみをインターンしてフラッシュメモリに配置し，プログラム実行中に生成される文字列には個別にメモリを割り当てる方式を提案した（5章）．インターンしている文字列同士の比較には，文字列のアドレスによる比較を行う．個別にメモリを割り当てた文字列との比較は，文字列の内容による比較を行う．この方式ではコンパイル時にインターンするので，実行時にハッシュ表をRAMに配置する必要がなくなる．その代わり，個別にメモリを割り当てた文字列を比較する際は，アドレスによる比較ができなくなるため比較が遅くなる．

提案する文字列管理方式では，文字列の比較方法を判断する処理が頻繁に行われるようになり，そこにオーバヘッドがかかる可能性がある．実行時間を増加させないためにも，なるべく文字列の比較方法を判断する処理は除外したい．JavaScriptプログラム中のどこで文字列の比較を行っているかは，コンパイル時に知ることができる．そこで，コンパイル時に文字列の比較方法を決定する方法を考案した（5.2.1節）．また，オブジェクトのプロパティ名に動的に生成される文字列が使用されることはほとんどない．そこで，キャッシュに保存

されたプロパティであるか判定する際に，文字列のアドレスによる比較のみを行う方法を考案した（5.2.2 節）．

提案方式の効果を確かめるため，提案方式を実装しベンチマークプログラムに適用した．その結果，文字列を管理するハッシュ表を無くしたことで，今回の実験の範囲では RAM の空き領域が 20 KiB，JavaScript のオブジェクトを格納するヒープのサイズが最大で 2.4 KiB 増加した．一方，実行時間が既存の eJSVM と比べて最大で 86 %，平均で 6 %増加した．コンパイル時に文字列の比較方法を決定する方法を用いると，既存の eJSVM と比べて最大で 86 %，平均で 2 %減少した．キャッシュされたプロパティとの比較で文字列のアドレスによる比較のみを行う方法を用いた場合は，既存の eJSVM と比べて最大で 83 %，平均は既存の eJSVM と同程度だった．この結果から，提案方式と文字列の比較方法を判断する処理を除外する方法を組み合わせることで，一部を除き既存の eJSVM とほぼ変わらない実行時間になることがわかった．

第 2 章

組み込みシステム向け JavaScript 処理系 eJS

2.1 eJSVM

eJS は，組み込みシステムのアプリケーション開発を JavaScript で行うために開発を進めている JavaScript 処理系である．eJS は，仮想機械（VM）である eJSVM を提供している．eJSVM は，組み込みシステムで実行することを想定して作られた小型の JavaScript VM であり，100 KB 程度のメモリで実行することができる．

eJSVM は，eJS が提供するコンパイラ（以下，eJSC）で生成されたバイトコードを実行するインタプリタである．eJSVM では UNIX 系 OS 上で実行するモードと，組み込みシステム向け OS のライブラリとリンクして，マイコン上で実行するモード [2] がある．UNIX 上で実行する場合，生成されたバイトコードはファイルに記述され，ファイルシステムから読み取り実行する．

組み込みシステムで実行する場合，バイトコードはコンパイル時に eJSVM に直接埋め込まれる．この時，バイトコードから参照されるデータも一緒に埋め込まれ，フラッシュメモリに配置される．組み込みシステムの中にはファイルシステムが存在しないものがあるため，ファイルシステムの有無に関わらず実行できるような形式をとっている．

アプリケーションに合わせた VM を生成するために，eJSVM では開発者から与えられた各種仕様に基づいて VM のソースコードを生成し，それをコンパイルして eJSVM を生成する．eJSVM 生成過程を図 2.1 に示す．開発者から与えられたデータ型仕様からデータ型

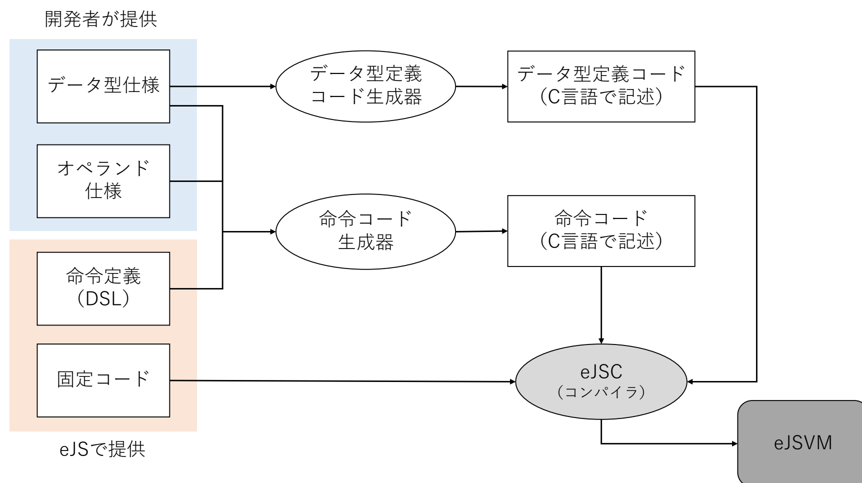

図 2.1　eJSVM の生成過程

定義コードが，データ型仕様とオペランド仕様，eJS が提供する命令定義から命令コードが生成される．これらを，eJSVM 共通の固定コードと一緒にコンパイルすることで eJSVM が得られる．

例えば，数値のみの加算を行うアプリケーションについて考える．加算は `add` 命令で行われるが，`add` 命令内ではオペランドのデータ型に応じて異なる処理が行われる．例えば,「1 + 2 = 3」のように，数値同士の加算の場合は通常の加算を行う.「‘a’ + ‘b’ = ‘ab’」のように，文字列同士の加算の場合は文字列の結合を行う．今回のアプリケーションでは，数値のみの加算を行うため，数値同士以外の処理は必要がない．そこで，開発者は図 2.2 のように，オペランドのデータ型に応じて各処理が必要か必要でないか指定することができる．この記述を「オペランド仕様」と呼ぶ．図 2.2 では，`add` 命令の第二オペランドと第三オペランドに eJSVM のデータ型である「`fixnum`」か「`flonum`」が与えられた場合，処理を許容する（`accept`）．それ以外のデータ型が与えられた場合は，エラーとする（`error`）．

オペランド仕様とデータ型仕様，命令定義を命令コード生成器に与えることで，そのアプリケーションに必要な処理のみを行う命令コードが生成される．生成された命令コードとデータ型定義コード，固定コードを eJSC でコンパイルすることで，不必要な処理を除外した小型な VM の生成を可能とする．

```
1 | add (−,fixnum,fixnum) accept
2 | add (−,flonum,flonum) accept
3 | add (−,_,_) error
```

図 2.2　オペランド仕様

2.2　JavaScript ヒープ

JavaScript ヒープ（以下，JS ヒープ）は，JavaScript プログラムがオブジェクトを格納するメモリ領域である．JavaScript ヒープはガベージコレクション（以下，GC）で管理されており，GC は使用されていないオブジェクトを JS ヒープから回収する．eJSVM では複数の GC を実装しているが，本研究ではマークスイープ GC を使用する．

2.3　インラインキャッシュ

インラインキャッシュ[3, 4] は，オブジェクトのプロパティ検索を高速化する手法である．JavaScript では，文字列式の値をプロパティ名としてオブジェクトのプロパティにアクセスすることができる．その場合，実行時までどのプロパティにアクセスするかわからない．さらに，オブジェクトのどのオフセットにどのプロパティ値が格納されているかわからない．そのため，eJSVM では実行時にオブジェクトのプロパティ名からプロパティのオフセットを検索するようになっており，オフセットはオブジェクトが持つ Hidden クラス [5] と呼ばれるハッシュ表で管理されている．

一方，プログラム中のある命令においてアクセスされるオブジェクトの Hidden クラスやプロパティ名は，繰り返し命令が実行されても変化しないことが多いことが知られている．インラインキャッシュは，プログラム中の各プロパティアクセス命令に対して割り当てる記憶領域であり，その命令において最後にアクセスしたオブジェクトの Hidden クラスとプロパティ名，そのプロパティのオフセットを保存する．次回のアクセスにおいて，Hidden クラスとプロパティ名が保存されているものと一致していれば，Hidden クラス内を検索せず，

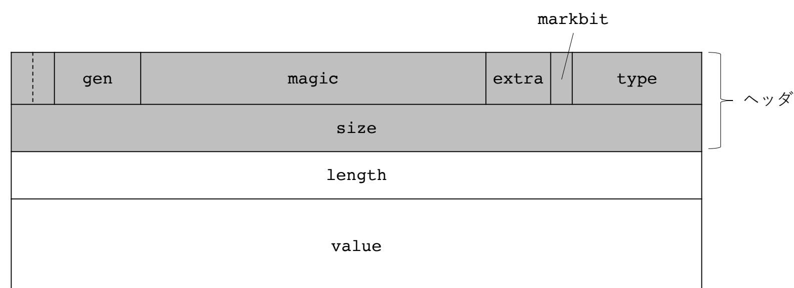

図 2.3　文字列オブジェクトのデータ構造

保存されているオフセットを使用する．これにより，Hidden クラス内を検索を減らし，プロパティへのアクセスを高速化する．

2.4　文字列

2.4.1　文字列オブジェクト

eJSVM では，JavaScript の文字列を表すために文字列の長さや文字列本体の情報を持つデータ構造を生成している．以下では，このデータ構造を文字列オブジェクトと呼ぶ．文字列オブジェクトのデータ構造を図 2.3 に示す．eJSVM で生成されるオブジェクトは共通のヘッダを持っており，ヘッダにはデータ型を示す `type` フィールドや，オブジェクトの大きさを示す `size` フィールドなどがある．文字列オブジェクトは，ヘッダと文字列の長さを格納する `length` フィールド，文字列本体を格納する `value` フィールドで構成されている．

2.4.2　静的文字列と動的文字列

eJSVM では，プログラム実行前に生成される文字列（以下，静的文字列）はフラッシュメモリに配置している．静的文字列には以下のものがある．

- プログラム中の文字列定数
- プロパティ名
- グローバル変数名
- eJSVM 内部で生成する文字列

静的文字列の文字列オブジェクトは図 2.4 の形で生成される．各関数が使用している静的文字列の文字列オブジェクトは，関数が持つ定数テーブルに格納されている．例えば，関数 foo では関数名 foo，文字列 Hello，文字列 World が使用されているため，これらの文字列オブジェクトへのアドレスが定数テーブルに格納されている．また，eJSVM 起動時に eJSVM 内部で生成される文字列は関数が持つ定数テーブルとは別の定数テーブル（以下，グローバル定数テーブル）が文字列オブジェクトへのアドレスを保持している．定数テーブルに格納されている文字列は，コンパイル時にインターンしているため，同じ文字列は一つの文字列オブジェクトを共有している．図 2.4 では，関数 foo と関数 bar は文字列オブジェクト Hello を，関数 bar とグローバル定数テーブルは文字列オブジェクト Hi を共有している．

一方，プログラム実行中に生成される文字列（以下，動的文字列）は JS ヒープに配置される．動的文字列には以下のものがある．

- toString などプログラム実行中に文字列を生成するメソッド
- +演算子で結合した文字列

オブジェクトのプロパティに動的文字列が使用される場合もある．例えば，a['fo' + 'o'] はオブジェクト a のプロパティfoo を表しており,「a.foo」という記述と同じ意味になる．eJSVM のプロパティアクセス命令においては，前者は引数であるプロパティ名が動的文字列で与えられ，後者はプロパティ名が静的文字列で与えられる．

例を図 2.5 に示す．左は静的文字列の例を示している．関数名 main や print，文字列リテラル he や llo，プロパティ名 foo，toString メソッドが静的文字列に該当する．これらの文字列は，コンパイル時に文字列オブジェクトが生成される．一方，右は動的文字

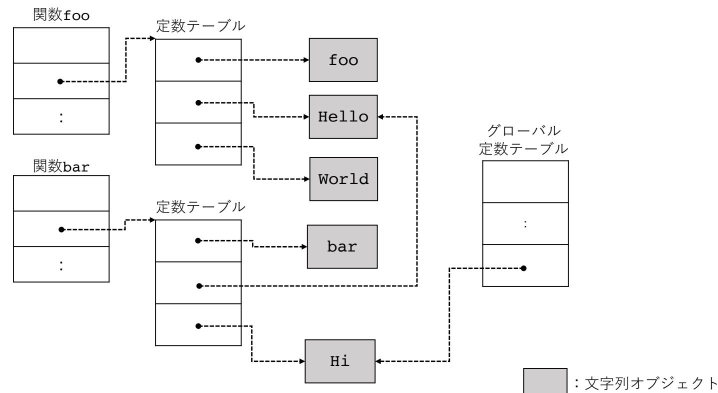

図 2.4　フラッシュメモリ中の文字列オブジェクト

```
function main() {
        var a = {foo:'he'};
        var b = 'llo';
        var c = 123;

        print(a.foo + b);
        print(c.toString());
}

main();
```

```
function main() {
        var a = {foo:'he'};
        var b = 'llo';
        var c = 123;

        print(a.foo + b);     → hello
        print(c.toString());  → 123
}

main();
```

青文字：静的文字列　　赤文字：動的文字列

図 2.5　静的文字列と動的文字列の例

列の例を示している．a.foo と変数 b を+演算子で結合することで動的文字列 hello が，c.toString() で動的文字列 123 がそれぞれ生成される．

2.4.3　eJSVM の文字列管理方式

eJSVM では，JavaScript プログラムの実行中，最初に文字列が出現した時に文字列オブジェクトを生成し（静的文字列は既に生成されているため除く），同じ文字列が出現した場

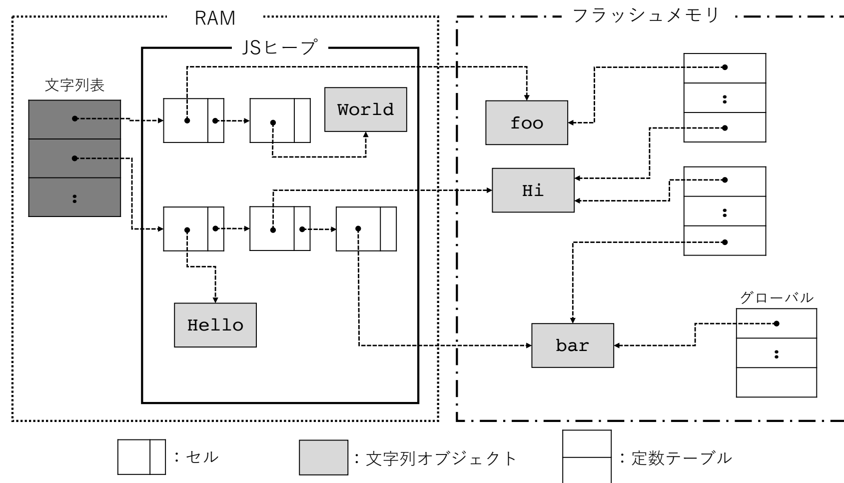

図 2.6　eJSVM の文字列管理方式

合は一つの文字列オブジェクトを共有するインターンを行っている．全体像を図 2.6 に示す．文字列をインターンするために，全ての文字列オブジェクトへのアドレスを保持するハッシュ表（以下，文字列表）が文字列を管理している．文字列表は外部ハッシュ表を採用しており，ハッシュ表本体と文字列オブジェクトへのアドレスを持ったセルの連結リストで構成される．eJSVM は，ある文字列 s を持つ文字列オブジェクトが必要になった時，次のステップで文字列オブジェクトを得る．

1. s のハッシュ値を計算
2. 文字列 s と同じ文字列を持つ文字列オブジェクトが文字列表に登録されているか調べる
3. 登録されている場合，その文字列オブジェクトへのアドレスを返す
4. 登録されていない場合，以下の処理を行う
   i. 文字列 s を持つ文字列オブジェクトを生成する
   ii. 生成した文字列オブジェクトを文字列表に登録する
   iii. 生成した文字列オブジェクトへのアドレスを返す

文字列をインターンすることで，以下のメリットが得られる．

- 文字列オブジェクトを共有することで，同じ文字列が複数生成される時に空間効率がよ

くなる

- アドレスを比較することで，同じ文字列かどうかを高速に判定できる

JavaScript において文字列オブジェクトは不変であり，一度生成された文字列オブジェクトが変更されることはない．そのため，同じ文字列は一つの文字列オブジェクトに格納しそれを共有することで，文字列オブジェクトに割り当てるメモリを節約することができる．

一方で，以下のようなデメリットもある．

- 文字列表が RAM を圧迫する
- プログラム実行中に文字列が出現する度に，文字列表に登録されているか調べる必要がある
- 文字列表から文字列オブジェクトへの参照は弱参照になっているため，その処理のために GC の時間がかかる

文字列表本体は eJSVM 起動時に RAM に確保するが，必要十分な文字列表本体のサイズは事前に知ることができないため，現在の実装では 20 KiB としている．しかし，文字列オブジェクトがほとんど生成されない場合，無駄に RAM を消費することになる．また，文字列表のセルは JS ヒープに配置されており，文字列オブジェクトの数だけ生成される．そのため，多量に文字列オブジェクトが生成されると，その数だけセルも生成するため JS ヒープが圧迫される．

2.4.4　文字列に対する処理

eJSVM では，二つの文字列が等しいかどうかを判定する処理は頻繁に行われる．eJSVM 内部では，文字列の比較は主に以下の箇所で行われる．

- ==演算子と===演算子に対応する命令
- オブジェクトのプロパティ検索

2.4　文字列

文字列の内容に対しての処理は，文字列の出力や結合，`String.prototype.split` 関数などの組み込み関数など限られた箇所でしか行われない．

第 3 章

Mbed

eJS や本研究の提案方式は特定の組み込みシステムに依存しないが，提案方式の実装と評価実験は本章で述べる組み込みシステムを使用する．本章ではこの組み込みシステムについて説明する．

3.1 Mbed の全体像

Mbed[6] は ARM 社が開発した IoT デバイスの開発環境の名称であり，OS や API が無償で提供されている．Mbed の OS は，OS の機能を提供するライブラリ（OS ライブラリ）である．

3.2 FRDM-K64F

本研究で使用する Mbed に対応したデバイス（以下，Mbed デバイス）は，NXP Semiconductors の Freedom 開発プラットフォームである FRDM-K64F[*1] である．実際のデバイスの写真を図 3.1 に示す．仕様は表 3.1 の通りである．FRDM-K64F は，RAM が SRAM_L と SRAM_U の 2 つの領域に分かれており，合わせて 256 KB となる．

[*1] https://www.mouser.jp/new/nxp-semiconductors/freescale-frdm-k64f-dev-board/

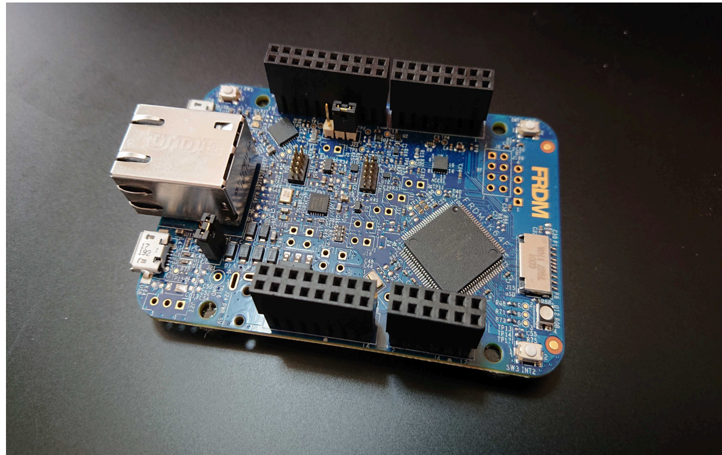

図 3.1　FRDM-K64F

表 3.1　FRDM-K64F の性能

CPU コア	Cortex-M4F（32 ビット）
フラッシュメモリ	1 MB
SRAM_L	64 KB
SRAM_U	192 KB

3.3　メモリマップ

SRAM_L のメモリマップを図 3.2，SRAM_U のメモリマップを図 3.3 に示す．SRAM_L の`.heap_0` セクションは，C 言語の標準ライブラリが `malloc` でメモリを割り付ける領域である．eJSVM では，VM の起動時に以下のデータを割り当てている．

- JavaScript プログラム実行のためのスタック
- インラインキャッシュの領域
- 文字列表

SRAM_U にある `.bss` セクションは未初期化のデータが，`.data` セクションは初期化済みのデータが配置される．`.stack` セクションは，Mbed OS が使用するスタック領域である．`.heap` セクションは，`.heap_0` セクションと同様に C 言語の標準ライブラリが `malloc` で

3.3 メモリマップ

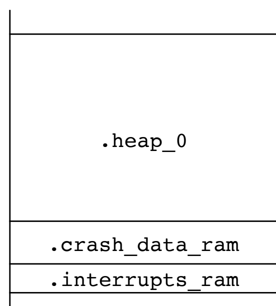

図 3.2 SRAM_L

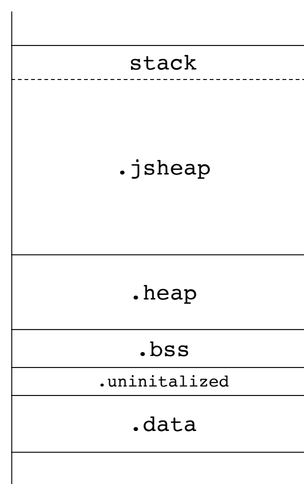

図 3.3 SRAM_U

メモリを割り当てる領域である．.heap_0 セクションにメモリを割り当てることができなくなったら，.heap セクションを使用する．.jsheap セクションは，JS ヒープ専用の領域である．

第 4 章

既存の文字列管理方式

4.1 個別生成方式

eJSVM では，2.4.3 節で述べたように文字列をインターンしているが，インターンせず，同じ文字列であっても個別に文字列オブジェクトを生成する方式もある．この方式を個別生成方式と呼ぶ．この方式では，文字列表を用いないため文字列表が占めていた RAM 領域が不要になる．また，ハッシュ値の計算も無くなるため文字列オブジェクトの生成が高速になるメリットがある．しかし，同じ文字列でも個別に文字列オブジェクトが生成されているため，文字列を比較する際に文字列の内容で比較する必要があり，比較に時間がかかるデメリットもある．JS ヒープに関しては，文字列オブジェクトへのアドレスを持つセルが無くなる一方で，同じ文字列の文字列オブジェクトが複数生成されるようになる．

4.1.1 予備実験

文字列をインターンする既存の eJSVM（以下，インターンする方式）と個別生成方式で実行時間にどの程度差が出るかを調べるため，個別生成方式を eJSVM に実装し，実行時間を計測した．実行環境の詳細は 6 章で述べる実験と同じである．ベンチマークは，sunSpider ベンチマーク [*1] の一部と，AWFY ベンチマーク [7] の一部を eJSVM 用に改変したものを使用した．それに加え，動的文字列を多く生成するプログラムを二つ（`CaesarCipher.js` と `StringRotation.js`），プロパティ名に動的文字列を使用するプロ

[*1] https://webkit.org/perf/sunspider/sunspider.html

4.1　個別生成方式

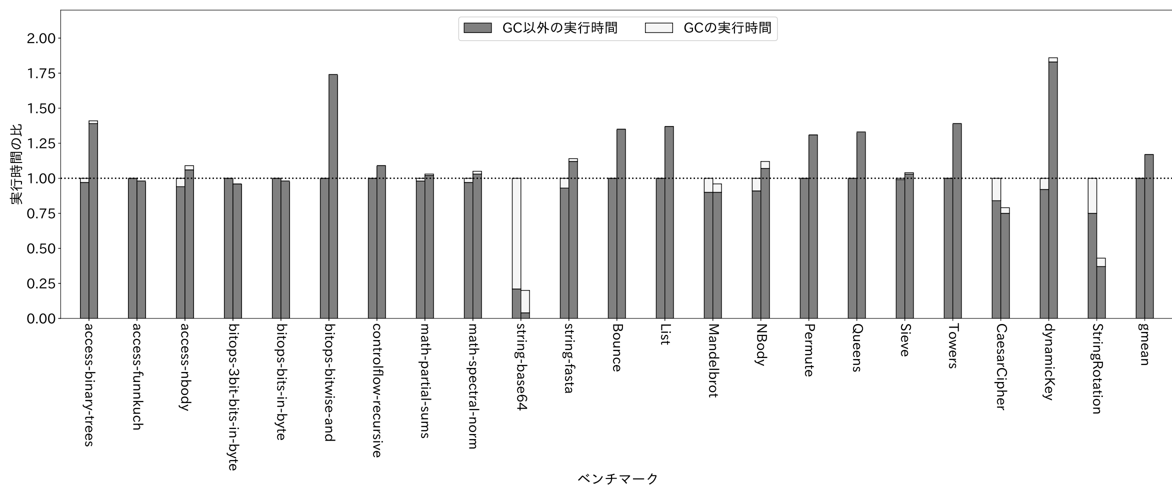

図 4.1　インターンする方式と個別生成方式の実行時間の比

グラムを一つ（dynamicKey.js）用意した．これらのプログラムの詳細も 6 章で説明する．

結果を図 4.1 に示す．ベンチマークごとに，左がインターンする方式，右が個別生成方式であり，インターンする方式の実行時間を 1 とする比で表している．インターンする方式と比べて，個別生成方式のほうが極端に傾向が異なる string-base64 と人工的なプログラムである CaesarCipher，dynamicKey，StringRotation を除いて平均で 17 %増加した．また，22 個中 10 個のベンチマークで 10 %以上実行時間が長くなっており，特に dynamicKey は 86 %増加していた．一方，実行時間が 10 %以上短くなったベンチマークは string-base64 と CaesarCipher，StringRotation の 3 個だった．これらのプログラムは，動的文字列を多く生成する．個別生成方式では，文字列オブジェクトが文字列表に登録されているかどうかを調べる必要が無くなったため，実行時間が短くなったと考える．dynamicKey も動的文字列を多く生成するが，プロパティへのアクセスが頻繁に行われているため，そこにオーバヘッドがかかっていると考える．また，個別生成方式は GC の実行時間も短くなっている．これは，文字列表のセルが無くなったことで JS ヒープの空き領域が増加し，他のオブジェクトがその領域を使用できるようになったことが原因であると考える．

4.1.2　文字列の比較回数の調査

4.1.1 節の実験から，個別生成方式の方が実行時間が長くなるベンチマークの方が多かった．インターンする方式と個別生成方式の違いの一つは，文字列の比較方法が異なることである．文字列の内容による比較は，文字列のアドレスによる比較より遅いため実行時間が長くなったと考える．これを確認するために，各プログラムの実行中に何回文字列の比較を行っているか調べた．また，インターンする方式を実装している既存の eJSVM に対し，文字列の比較方法だけを文字列の内容による比較を行うように変更し，比較方法を変更したことで実行時間がどの程度増加したか調べた．

1 秒当たりの比較回数と，その比較がどこで行われたかの内訳を図 4.2 に示す．また，文字列の比較回数と実行時間の比の関係を表す散布図を図 4.3 に示す．なお，この図で言う実行時間の比は，既存の eJSVM の実行時間を 1 とした時の文字列の内容による比較を行う変更を行った eJSVM の実行時間である．図 4.2 に示す結果から，今回の実験では文字列の比較はほとんどがプロパティ検索で行われており，比較を行う命令の実行回数やそれ以外の箇所における文字列の比較は少ないことがわかった．また，図 4.1 と照らし合わせると，1 秒当たりの文字列の比較回数が 10 万回以上だったベンチマークでは，一部を除いて実行時間が 10％以上増加していた．図 4.3 を見ても，1 秒当たりの比較回数と実行時間の増加の割合がほぼ比例していることがわかる．このことから，個別生成方式で実行時間が遅い原因の一つは，文字列の比較であると考える．

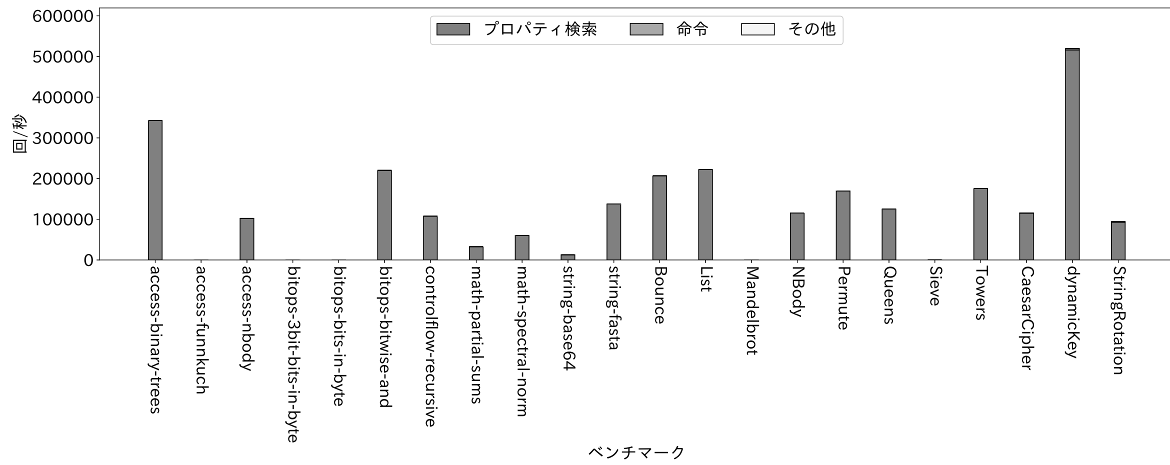

図 4.2 1 秒当たりの文字列の比較回数

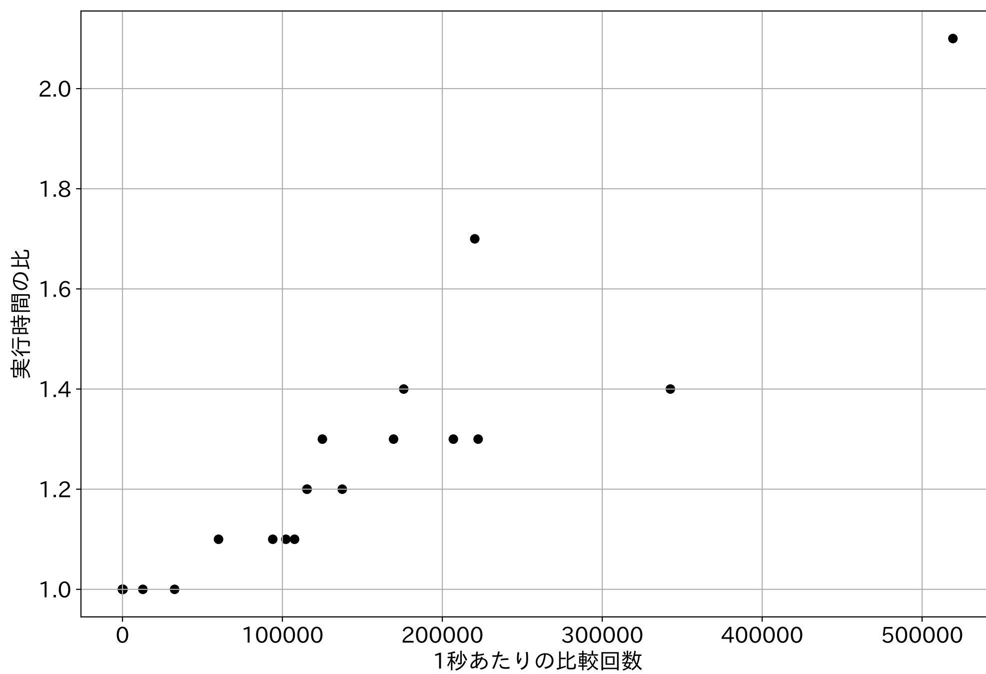

図 4.3 文字列の比較回数と実行時間の比の関係

第 5 章

新しい文字列管理方式の提案

5.1　ハイブリッド文字列管理方式

4.1 章で述べた個別生成方式では，文字列表が不要になったことで JS ヒープの空き領域が増加し，GC の実行時間も減少した．一方で，文字列の比較に時間がかかるため，ほとんどのベンチマークで全体の実行時間は増加した．本節では，文字列表を使用せずに文字列の比較にかかる時間を減少させる方法について考える．

eJSVM では，静的文字列はコンパイル時にインターンし，フラッシュメモリに配置している．そのため，静的文字列同士であれば文字列のアドレスによる比較が可能である．プログラム実行中に現れる文字列はほとんどが静的文字列であり，動的文字列が生成される頻度は少ない．

そこで，動的文字列をインターンしないことで文字列表を不要としつつ，静的文字列同士の比較の時だけ文字列のアドレスによる比較を行う方式を提案する．この方式をハイブリッド文字列管理方式と呼ぶ．全体像を図 5.1 に示す．提案方式では，4.1 節と同じように文字列表を用いない文字列管理を行う．静的文字列は事前にインターンし，動的文字列は個別にメモリを割り当てる．例えば，文字列 `foo` を持つ文字列オブジェクトは JS ヒープに二つ，フラッシュメモリに一つ生成されている．

提案方式では，以下のメリットが考えられる．

- 文字列表を用いないため，文字列表が占めていた JS ヒープや RAM の領域を他のデータが使用することができる

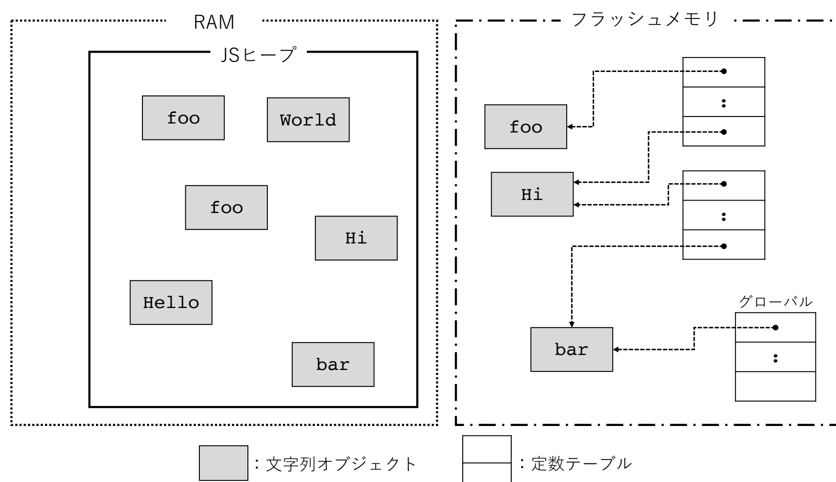

図 5.1　ハイブリッド文字列管理方式

- 文字列表の探索時間が無くなる
- 静的文字列同士の比較をアドレスで行うことで，個別生成方式より比較が高速化される

一方で，以下のデメリットが考えられる．

- 動的文字列同士または動的文字列と静的文字列の比較が多いと，文字列の内容による比較が増加するためインターンする方式より実行時間が増加する
- フラッシュメモリへのアクセスを行うことによるオーバヘッドが発生する

5.1.1　動的な文字列比較

提案方式では，以下の方法で文字列の比較を行う．文字列の比較方法の決定は，プログラム実行中に行われる．この手法を動的な文字列比較と呼ぶ．

1. 静的文字列同士の比較の場合，文字列のアドレスによる比較を行う
2. 動的文字列同士または動的文字列と静的文字列比較の場合，文字列の内容による比較を行う

```
1 string_compare(str1, str2) {
2         if (in_jsheap(str1) || in_jsheap(str2)) {
3                 return strcmp(str1, str2) == 0 ? 1 : 0;
4         } else {
5                 return str1 == str2;
6         }
7 }
```

図 5.2　動的な文字列比較

擬似コードを図 5.2 に示す．比較対象の 2 つの文字列のうち，どちらか一方でも JS ヒープに文字列オブジェクトが生成されていれば，文字列の内容による比較を行う．文字列の内容による比較は，eJSVM 内部で `strcmp` 関数を使用する．両方の文字列オブジェクトが JS ヒープに生成されていない，つまりフラッシュメモリに配置されている場合は，静的文字列であるため文字列のアドレスによる比較を行う．

5.2　動的な文字列比較の高速化

5.1.1 節で述べた動的な文字列比較（図 5.2）では，文字列の比較を行う度に静的文字列同士の比較であるかどうかを調べる必要があり，このことが実行時間のオーバヘッドになる可能性がある．そのため，文字列のアドレスによる比較を行うか内容による比較を行うかを判断する処理が必要のない場面では，その処理を除外したい．

最も文字列の比較回数が多いプロパティ検索では，オブジェクトの Hidden クラスとプロパティ名からキャッシュされているオフセットを検索する．そのため，キャッシュに保存されたプロパティ名との比較は頻繁に行われる．プロパティ名はほとんど静的文字列が使用されるため，実際には文字列のアドレスによる比較だけでよい場面が多々ある．そこで，キャッシュに保存されたプロパティ名と一致するかどうか調べる処理について，２種類の高速化法を考案する．

5.2.1　静的な文字列比較

プロパティへのアクセスを行っている箇所は，コンパイル時にバイトコードが生成される過程で知ることができる．プロパティ名に必ず静的文字列が使用されることが事前に分かれば，文字列のアドレスによる比較のみを行うだけでよい．

そこで，JavaScript プログラムのコンパイル時に比較方法を決定する．この手法を静的な文字列比較と呼ぶ．コンパイル時に，プロパティへのアクセスを行っている箇所でプロパティ名にあたる文字列が静的文字列と動的文字列のどちらであるか調べ，静的文字列のみが使用されている場合は文字列のアドレスによる比較のみを行うような命令を生成する．

プロパティ名に静的文字列を使用しているかどうかの判定は，コンパイラ最適化の一つである定数伝播 [8] を応用し eJSC コンパイラに実装する．定数伝播は，ある変数 a について，値が定数 c であるとわかっている場合は，変数 a を定数 c に置き換える手法である．

静的な文字列比較の全体像を図 5.3 に示す．命令の内部でプロパティ検索を行っているのは以下の 4 つである．これらの命令それぞれについて，同じことを行うが文字列の比較の際はアドレスによる比較のみを行う命令を用意する．それらの特殊化した命令のことを static 系命令と呼ぶ．static 系命令はキャッシュに保存されているプロパティ名との比較の時に文字列のアドレスによる比較のみを行う．一方，通常の命令は動的な文字列比較を行う．static 系命令の各オペランドには，表 5.1 に示すものが与えられる．

- `getprop` 命令：プロパティ値の取得
- `setprop` 命令：プロパティ値のセット
- `getglobal` 命令：グローバル変数などの取得
- `setglobal` 命令：グローバル変数などのセット

プロパティ名に静的文字列を使用しているかどうかは，以下の手順で調べる．

1. プロパティ名にあたるオペランドを評価する
2. オペランドが定数の場合，静的文字列であるため static 系命令に置き換える

表 5.1 各 static 系命令の引数

命令	第一オペランド	第二オペランド	第三オペランド
getprop	戻り値	オブジェクト	プロパティ名
setprop	オブジェクト	プロパティ名	プロパティ値
getglobal	戻り値	プロパティ名	なし
setglobal	プロパティ名	プロパティ値	なし

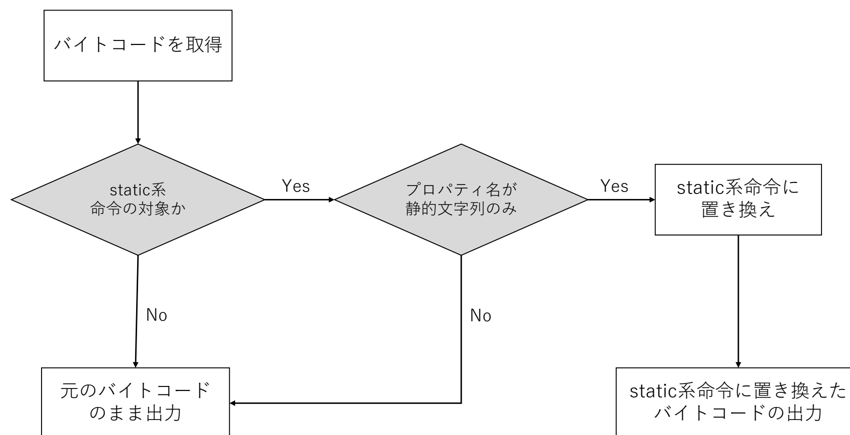

図 5.3 static 系命令への置き換え

3. オペランドのがレジスタ r の場合，レジスタ r の到達定義（ある地点からレジスタ r に到達する命令）を全て取得する
4. 取得した定義が全て以下の条件のいずれかを満たすなら，static 系命令に置き換える
   条件 1. 定数をロードする命令である
   条件 2. move 命令や getprop 命令など値を取得する命令であり，これらの命令に到達する定義に定数をロードする命令などがある（これらの命令のオペランドが静的文字列であるかどうかは，この手順を再帰的に適用して調べる）

「getprop reg1 reg2 reg3」というバイトコードを例に考える．例は，getprop 命令であるため，static 系命令の対象である．そこで，getprop 命令の第三オペランドに与えら

れるプロパティ名が常に静的文字列が使用されているか調べる．第三オペランドはレジスタ（reg3）であるため，reg3 に到達する定義を調べる．ここで，reg3 には定数をロードする命令だけが到達していることがわかったとする．定数は静的文字列であるため，この命令を実行する際，reg3 は静的文字列だけを参照することがわかる．よって,「getpropstatic reg1 reg2 reg3」に置き換えることができる．

プロパティへのアクセスに使用されるプロパティ名はほとんどの場合静的文字列であるため，プロパティアクセス命令のほとんどが static 系命令に置き換わると考える．これにより，最も頻繁に文字列の比較を行う箇所であるプロパティ検索でのオーバヘッドが削減できると考える．

5.2.2　楽観的な文字列比較

プロパティ名は，実際には動的文字列を使用することはほとんどなく，キャッシュされたプロパティ名と一致するかどうか調べる処理では静的文字列同士の比較が多くなる．静的文字列は事前にインターンしてあり，アクセスするオブジェクトのプロパティ名がキャッシュに保存されているプロパティ名と同じ文字列を使用していれば，同じ文字列オブジェクトへのアドレスを持っていることになる．

そこで,「ほとんどの場合，プロパティ名には静的文字列が使用される」と仮定して，キャッシュに保存されているプロパティ名との比較は，常に文字列のアドレスによる比較で行う．この手法を楽観的な比較と呼ぶ．プロパティ名が静的文字列でも動的文字列でも，キャッシュに保存されているプロパティ名との比較では，文字列のアドレスによる比較を行うようになる．

擬似コードを図 5.4 に示す．楽観的な文字列比較は，プロパティ名が同じ文字列オブジェクトを参照している場合は，アドレスによる比較により高速な比較ができる．一方，キャッシュに保存されているプロパティ名と違う文字列オブジェクトを参照している場合はキャッシュミスになり，Hidden クラスからオフセットを検索する．キャッシュに保存されているプロパティ名と同じ文字列であっても，参照先の文字列オブジェクトが異なればキャッシュミ

スになる．ただし，キャッシュミスになってもオブジェクトの Hidden クラスを検索するため，実行時間は遅くなるが間違ったプロパティ値が取得されることはない．

動的文字列は個別にメモリを割り当てており，仮に値が等しくても同じ文字列オブジェクトを参照していることは少ない（一度生成された動的文字列の文字列オブジェクトを持ち続けていることはほぼない）．そのため，プロパティ名に動的文字列が使用されている場合，キャッシュミスの頻度が高くなることが考えられる．これにより，実行時間が長くなってしまう可能性がある．

静的な文字列比較と楽観的な文字列比較の違いは，静的な文字列比較はコンパイル時に比較方法を決める．プロパティアクセス命令は，キャッシュに保存されているプロパティ名との比較時にアドレスによる比較のみを行う static 系命令か，動的な文字列比較を行う通常の命令になる．そのため，参照先の文字列オブジェクトが異なっても，同じ文字列を使用している場合はキャッシュに保存されているオフセットを使用できる．一方，楽観的な文字列比較はプロパティ名が静的文字列でも動的文字列でもアドレスによる比較を行う．そのため，同じ文字列を使用していても，参照先の文字列オブジェクトが異なる場合は Hidden クラスを検索することになる．

```
// obj = オブジェクト, prop = プロパティ
プロパティ検索(obj, prop) {
      // プログラムカウンタを引数としてキャッシュを取得
      InlineCache *ic = GET_INLINE_CACHE(pc);
      HiddenClass hc = GET_HIDDEN_CLASS(obj);

      if (ic->Hiddenクラス == hc &&
          ic->プロパティ名 == prop) {
           // キャッシュされているオフセットを使用してプロパティ値を取得

      } else {
           // プロパティ名をキーとして，hcからオフセットを取得
           // オフセットを使用してプロパティ値を取得
           // キャッシュにhcとpropとプロパティのオフセットを保存
      }
}
```

図 5.4　楽観的な文字列比

第 6 章

評価

文字列表を無くしたことで，RAM や JS ヒープの空き領域が増加したかどうか調査する．また，ハイブリッド文字列管理方式の実行時間は，インターンする方式と比べてどのぐらい差があるか比較する．実行環境は表 6.1 の通りである．

表 6.1 実験環境

デバイス	FRDM-K64F
OS	Mbed OS 6
ライブラリ	Newlib 3.1.0
コンパイラ	GCC_ARM

6.1 ベンチマーク

本実験で使用したベンチマークは，4.1.1 節で使用したプログラムと同じ sunSpider ベンチマークと AWFY ベンチマークである．また，動的文字列を多く生成する以下の 3 つのプログラムを作成した．

- `CaesarCipher.js`
- `dynamicKey.js`
- `StringRotation.js`

`CaesarCipher.js` は，大文字と小文字のアルファベット 10000 文字に対して，暗号方式の一つであるシーザー暗号を適用するプログラムである．プログラム中では，暗号化と復号を

行っており，暗号化または復号した文字列と元の文字列を 20 文字ごとに比較している．

`dynamicKey.js` は，プロパティ名に動的文字列を使用し，さまざまな方法でオブジェクトのプロパティに頻繁にアクセスするプログラムである．例えば，プロパティを次々に追加したり，プロパティ値を取得したりする．プロパティ名は文字列 `Key` とループ変数 `i` の値を結合した「`['Key' + i]`」の形になる．

`StringRotation.js` は，大文字と小文字のアルファベット 10000 文字を 10 文字ずつ区切り文字列を反転させる．反転した文字列は元の文字列と比較を行う．

6.2 RAM の使用状況

文字列表が無くなったことを確かめるために，eJSVM 起動直後の `.heap_0` セクションと `.heap` セクションに割り当てているデータのサイズを調査した．なお，`.heap_0` セクションのサイズは 64,356 B，`.heap` セクションのサイズは 43,440 B である（計 107,796 B）．結果を図 6.1 に示す．上がインターンする方式で，下がハイブリッド文字列管理方式である．空き領域となっている領域は，ネイティブ関数やそこから呼び出される Mbed のライブラリ OS の API などが利用できる領域である．インターンする方式と比べて，ハイブリッド文字列管理方式では全てのベンチマークで空き領域が増加していることが確認された．特に，インターンする方式では `NBody` は空き領域が最大で 19.5 KiB しかなかったが，ハイブリッド文字列管理方式では空き領域が 39.5 KiB 増加した．これは，文字列表が使用していた 20 KiB が空いたことによる効果であると考える．

6.3 JS ヒープの空き領域

文字列表のセルを生成しなくなったことで，JS ヒープの空き領域が増加したかどうかを確認するため，eJSVM 起動直後における JS ヒープの空き領域を，インターンする方式とハイブリッド文字列管理方式で比較した．

最も JS ヒープの空き領域が増加した `NBody` に対する結果を図 6.2 に示す．インターン

6.3　JS ヒープの空き領域

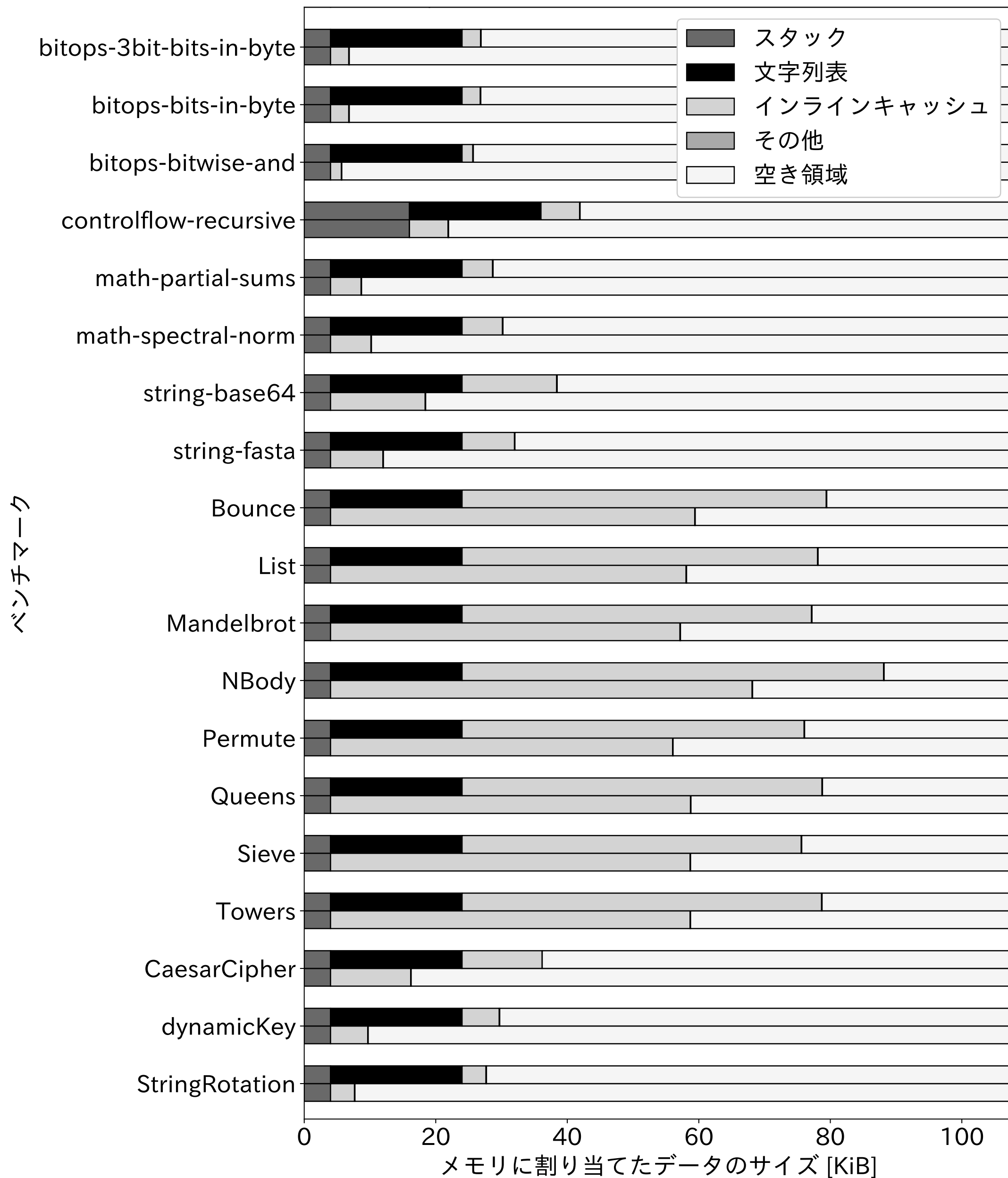

図 6.1　.heap_0 セクションと.heap セクションに割り当てているデータの総計

する方式の空き領域は 121,496 B，ハイブリッド文字列管理方式の空き領域は 123,884 B であった．インターンする方式と比べて，ハイブリッド文字列管理方式のほうが JS ヒープの空き領域が約 2.4 KiB 増加した．これは，eJSVM 起動時に生成される文字列表のセルが無くなったことによる効果であると言える．その他のベンチマークでも，約 1.1〜2.2 KiB の増加が確認された．

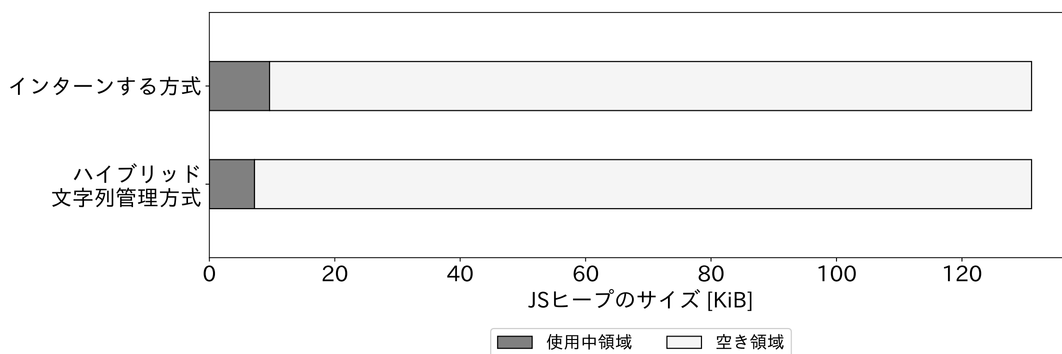

図 6.2 eJSVM 起動直後の JS ヒープの使用状況

6.4 実行時間

インターンする方式とハイブリッド文字列管理方式で実行時間に差があるかどうか比較を行った．また，ハイブリッド文字列管理方式に対して，静的な文字列比較と楽観的な文字列比較をそれぞれ実装した eJSVM の実行時間も比較する．

結果を図 6.3 に示す．ベンチマークごとに，左からインターンする方式，ハイブリッド文字列管理方式，静的な文字列比較を実装したハイブリッド文字列管理方式，楽観的な文字列比較を実装したハイブリッド文字列管理方式となっており，インターンする方式の実行時間を 1 とする比で表している．インターンする方式と比べて，極端に傾向が異なる `string-base64` と人工的なプログラムである `CaesarCipher`，`dynamicKey`，`StringRotation` を除いて実行時間が平均で 6 %増加した．個別生成方式では，平均で 17 %増加（4.1.1 節参照）しており，ハイブリッド文字列管理方式は一部を除いて個別生成方式よりも実行時間を短くすることができた．一方で，実行時間が 10 %以上長くなったプログラムは 9 個あり，個別生成方式の 10 個とほぼ同様だった．

静的な文字列比較を実装した場合は，インターンする方式と比べて実行時間が平均で 2 %減少し，楽観的な比較を実装した場合は，インターンする方式と同程度の実行時間になった．通常のハイブリッド文字列管理方式と比べると，一部を除いて実行時間は短くなっており，インターンする方式とほぼ同じ実行時間になっている．これは，動的な文字列比較を除

6.4 実行時間

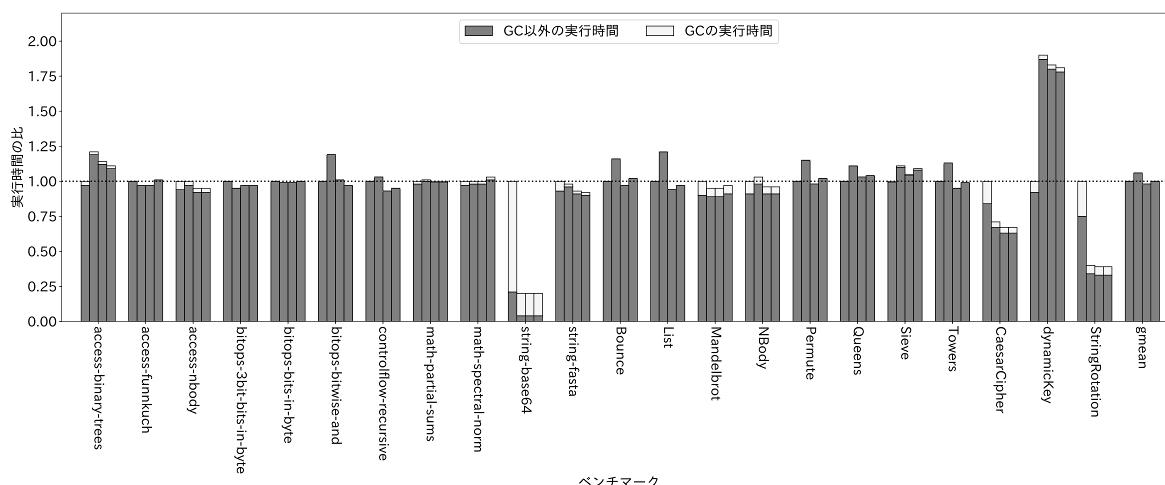

図 6.3 実行時間の比（左からインターンする方式，ハイブリッド文字列管理方式，静的な文字列比較，楽観的な文字列比較）

外したことで，実行時間を短くすることができたからであると考える．このことから，高速化法を導入していないハイブリッド文字列管理方式は，文字列の内容による比較を行うかアドレスによる比較を行うか，文字列の比較を判断する処理にオーバヘッドがかかっていたと考えられる．

一方で，dynamicKey はどの方式もインターンする方式と比べて 80％以上実行時間が増加していた．dynamicKey はプロパティ名に動的文字列が大量に使用されている．プロパティ名に動的文字列が使用されている場合，静的な文字列比較では通常の命令を使用することと，楽観的な文字列比較ではキャッシュミスの頻度が高くなる．このようなプログラムでは，静的な文字列比較と楽観的な文字列比較はほとんど効果がない．しかし，このプログラムは人工的な特殊な例であり，実際のプログラムでは，プロパティ名に動的文字列が使用されることはほとんどない．そのため，ハイブリッド文字列管理方式（静的な文字列比較と楽観的な文字列比較も含む）は全体的にはインターンする方式とほぼ変わらない実行時間になると考える．

第 7 章

関連研究

Kawachiya ら [9] は，文字列の実装方式として Java における文字列リテラルのインスタンス化を遅延する方式を提案している．Java の String クラスの典型的な実装は，文字列の情報を持つ `String` オブジェクトと文字列本体である `char` 配列オブジェクトで構成される．文字列リテラルがバイトコードで読み込まれた時，インターンされていなかった場合は `String` オブジェクトだけを生成する．`char` 配列オブジェクトの生成は，リテラルの値が実際に使用されるまで遅延させる．リテラルが実際に使用された時に `char` 配列オブジェクトを生成することで，未使用のリテラルによるメモリの消費を回避することが可能となる．また，文字列管理方式として StringGC を提案している．StringGC は，同じ内容を持つ `String` オブジェクトへの参照を，GC 時に同じ `String` オブジェクトを指すように正規化することで重複を除外することができる．GC で文字列オブジェクトを管理する方式は，新しい文字列管理方式として今後の参考になる．

Korsholm[10] は，著者らが作成した Java VM を使用して，Java 環境で配列などの定数データを配置している．ホスト初期化時に生成された定数データをマークしておき，マークされた定数データのフラッシュイメージを生成する．クロスコンパイラ環境でフラッシュイメージを一緒にコンパイルすることで，C 言語と同じ方式で実行可能ファイルが生成可能できる．eJS では，コンパイル時に静的なデータを配置しているが，著者らと同じ方式をとることで静的に生成される配列などのデータをフラッシュメモリに配置することができると考える．

Lisp 系の言語である Scheme 処理系でも，シンボル表についての工夫を行っている．Samuel[11] らの研究では，VM は Scheme のシンボル表を持たず，ライブラリ関数

`string->symbol` が持っている．関数 `string->symbol` や REPL を使用しない場合は，シンボル表が必要ないため除外することができる．また，Danny[12] らの研究では，Scheme のシンボル表のトリミングを行っている．Scheme プログラムの実行中に新しいシンボルが生成されることはほとんどない．動的にシンボルが生成された場合は，現在のシンボル表を拡張することで，必要なシンボルだけを持った表が生成される．この方法は，実行時にシンボルを生成しないプログラムで特に有効である．また，定数が構築された直後に定数の数に合わせてシンボル表を縮小することで，最小限のシンボル表が生成可能である．

第 8 章

まとめ

本研究では，組み込みシステム向け JavaScript 仮想機械における，RAM の利用可能領域を増やすために文字列表を用いないハイブリッド文字列管理方式を提案した．文字列の比較は，静的文字列同士は文字列のアドレスによる比較を行い，動的文字列と静的文字列，動的文字列同士の比較には文字列の内容による比較を行うようにした．また，プロパティ検索における文字列の比較を高速化させるために，文字列の比較方法を判断する処理を除外する方法を考案した．コンパイル時に比較方法を決定する静的な文字列比較と，プロパティ名には静的文字列が使用されると仮定してキャッシュされたプロパティと比較の際に文字列のアドレスによる比較を行う楽観的な比較をそれぞれハイブリッド文字列管理方式に実装し，高速化を図った．

提案方式を実装してベンチマークプログラムに適用した結果，RAM や JS ヒープの空き領域が増加したことを確認した．ハイブリッド文字列管理方式の実行時間は，既存の eJSVM と比べて平均で 6 %増加した．また，動的な文字列比較を除外する処理を行ったことで，静的な文字列比較では既存の eJSVM と比べて平均で 2 %減少し，楽観的な文字列比較では既存の eJSVM と同程度の実行時間になり，通常のハイブリッド文字列管理方式よりも実行時間を減少させることができた．特に，動的文字列が多く生成されるようなプログラムでは，文字列表を検索する必要がなくなり，GC の時間が大幅に減少したことで実行時間が最大で 81 %減少した．

一方で，実行時間が 80 %以上増加したプログラムが確認された．このプログラムはプロパティ名に動的文字列を大量に使用しており，このような特殊なプログラムの場合は，高速化の効果があまりないことがわかった．しかし，プロパティ名に動的文字列が使用される

ことは実際にはほとんどないため，全体的に見ればハイブリッド文字列管理方式は既存のeJSVMとほとんど変わらない実行時間になると考えられる．

謝辞

本研究を行うにあたり，終始ご指導いただいた高知工科大学の高田喜朗准教授，東京大学の鵜川始陽准教授並びに明治大学の岩崎英哉教授に深く感謝いたします．また，研究室の皆様をはじめ，本研究を支えてくださった皆様に感謝申し上げます．

参考文献

[1] Takafumi Kataoka, Tomoharu Ugawa, Hideya Iwasaki. A Framework for Constructing JavaScript Virtual Machines with Customized Datatype Representations, SAC'18, pages 1238–1247, 2018.

[2] 近森凪沙, JavaScript 処理系 eJS の Mbed マイコンへの移植と静的なデータのフラッシュメモリへの配置, 高知工科大学 学士学位論文, 2021.

[3] L.Peter Deutsch, Allan M.Schiffman. Efficient Implementation of the Smalltalk-80 System, POPL'84, pages 297–302, 1984.

[4] Zhefeng Wu, Zhe Sun, Kai Gong, Lingyun Chen, Bin Liao, Yihua jin. Hidden Inheritance: An Inline Caching Design for TypeScript Performance, OOPSLA'20, pages 1–29, 2020.

[5] Craig Chambers, David M.Ungar, and Elgin Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes, OOPSLA'89, pages 49–70, 1989.

[6] Mbed, `https://os.Mbed.com/`, 2020.

[7] Stefan Marr, Benoit Daloze, Hanspeter Mössenböck. Cross-Language Compiler Benchmarking: Are We Fast Yet?, DLS'16, pages 120–131, 2016.

[8] Andrew W.Appel. Modern Compiler Implementation in ML, 神林靖 (訳), 滝本宗宏 (訳). 最新コンパイラ構成技法, 株式会社 翔泳社.

[9] Kiyokuni Kawachiya, Kazunori Ogata, Tamiya Onodera. Analysis and Reduction of Memory Inefficiencies in Java Strings, OOPSLA'08, pages 385–402, 2008.

[10] Stephan Korsholm. Flash Memory in Embedded Java Programs, JTRES'11, pages 116–124, 2011.

[11] Samuel Yvon, Marc Feeley. A Small Scheme VM, Compiler, and REPL in 4K,

VMIL'21, pages 14–24, 2021.

[12] Danny Dubé, Marc Feeley. BIT: A Very Compact Scheme System for Microcontrollers, Higher-Order and Symbolic Computation, Volume 18, pages 271–298, 2005.