# ストレージストラテジーによる JavaScript オブジェクト配列のメモリ使用量の削減

永谷 龍彦　鵜川 始陽

センサの値を記録するような組み込みシステムで JavaScript を用いる場合，センサの値を記録するためのオブジェクトの配列のメモリ使用量が問題となる．本研究では，オブジェクトの配列のメモリ使用量を減らす手法を提案する．同じプロパティの集合を持つオブジェクトの配列に対して，オブジェクトへの参照の配列から，プロパティごとの配列の集合に変換する．そして，各プロパティの配列がプリミティブ型の配列になったとき，ストレージストラテジーという手法を適用することで，メモリ使用量を削減する．組み込み向けシステム向けの JavaScript 処理系 eJVSM に本手法を実装し，性能を評価した．オブジェクトの配列を単純なデータの記録に用いるプログラムでは，メモリ使用量を削減できることを確認した．その他のベンチマークプログラムでは，効果の薄い配列にまで手法が適用され，メモリ使用量は増加した．

## 1　はじめに

近年，JavaScript は文法の簡易さやライブラリの充実などの理由から，アプリケーション開発において広く用いられるようになった．しかし，組み込みシステム向けのソフトウェア開発においては，C や C++ といった低水準な言語が用いられることが多い．これは，組み込みシステムにおいては，使用できるメモリが限られているためである [6].

組み込みシステムが担う主な役割の一つは，センサからデータを取得し，より上位のシステムに送信することである [6]．このようなシステムでは，センサからのデータを一時的に保存するメモリが必要となる．JavaScript を含む多くのプログラミング言語において，このために用いられる言語機能は配列である．

図 1 は，センサから取得された温度や湿度を配列に記録し，一定数のデータが溜まったら上位システムに送信する C 言語のプログラムの例である．変数 `temperature` と `humidity` はそれぞれ `uint16_t`, `uint8_t` 型の配列のため，1 つの要素が消費するメモリは合計 3 バイトである．

一方，図 2 は，同じ処理を行う JavaScript のプログラムの例である．JavaScript などの高水準な言語では，性質の異なる複数の値がある場合，オブジェクトとしてまとめて扱うことが一般的である．このプログラムでは，変数 `array` は配列で，`temperature` と `humidity` という 2 つのプロパティを持つオブジェクトを要素として持っている．しかし，次の 2 つの理由から，図 1 で示した C 言語のプログラムよりも多くのメモリが必要になる．

一つは，オブジェクトの配列が，オブジェクトへのポインタの配列として表現されるためである．1 要素につきポインタ 1 つ分と，オブジェクトを管理するためのメタデータ分の領域が追加で必要となる．

もう一つは，JavaScript などの動的型付け言語では，オブジェクトのプロパティとして任意の型の値を格納できるようにするため，全ての値の大きさを揃える必要があるためである．このため，C 言語のプログラムであれば 1 バイトに収まっていた値も，最も大

---

Reducing Memory Footprint of JavaScript Object Arrays Using Storage Strategy

Tatsuhiko Nagaya, 東京大学情報理工学研究科, Graduate School of Information Science and Technology, the University of Tokyo.

Tomoharu Ugawa, 東京大学情報理工学研究科, Graduate School of Information Science and Technology, the University of Tokyo.

```c
uint16_t length = 0;
uint16_t temperature[3600];
unit8_t humidity[3600];
void onSensorChanged(temperature, humidity) {
  temperature[length] = temperature;
  humidity[length] = humidity;
  length++;
  if (length >= 3600) // Send data
    length = 0;
}
```

**図 1　センサの値を記録する C 言語のプログラム**

```javascript
var array = [];
function onSensorChanged(temperature, humidity)
{
  array.push({
    temperature: temperature,
    humidity: humidity
  });
  if (array.length >= 3600) // Send data
    array = [];
}
```

**図 2　センサの値を記録する JavaScript のプログラム**

きい値と同じ大きさのメモリを消費してしまう．

プリミティブ型の値からなる配列に対しては，ストレージストラテジーという手法を用いることで，メモリ使用量が削減できる [9]．ストレージストラテジーとは，配列に追加された要素の値から，自動的に最適なメモリレイアウトを決定する手法である．

本研究では，プリミティブ型の値だけでなく，オブジェクトを要素とする配列に対してもストレージストラテジーを適用することで，図 2 に示したようなオブジェクトの配列をデータの記録に用いるプログラムのメモリ使用量を削減する．具体的には，オブジェクトの配列をプロパティごとの配列に分解したうえで，各プロパティの配列に対してプリミティブ型の値に対するストレージストラテジーを適用する．これにより，全てのオブジェクトが同一の型の値を持つプロパティで，それがプリミティブ型であれば，そのプロパティの配列のメモリを削減できる．

提案手法を組み込みシステム向け JavaScript VM である eJSVM [7] に実装し，メモリ使用量と実行時間を計測した．その結果，図 2 のように，オブジェクトの配列が単純なデータの記憶に用いられている場合，メモリ使用量が削減できることが分かった．一方，その他のプログラムでは，メモリ使用量は増加した．

## 2　ストレージストラテジー

JavaScript などの動的型付け言語の VM は，静的型付け言語の VM と比べ，一つの値に対して多くのメモリを必要とする．これは，静的型付け言語ではコンパイル時に決定できる値の型情報を，動的型付け言語では実行時に保持しておく必要があるためである．また，動的型付け言語では，変数やプロパティに任意の型の値を格納できるようにするために，全ての値の大きさを揃えなければならない．このため，本来であれば小さな領域で表現できる値に対しても，最も大きな値の大きさに合わせてメモリを確保する必要がある．

```javascript
var array1 = []; // EmptyStrategy
array1.push(12);
array1.push(1.23); // NumberStrategy
array1.push(true); // JSValueStrategy
```

**図 3　ストレージストラテジーが遷移するプログラムの例**

しかし，動的型付け言語であっても，配列などのコレクションには，単一のデータ型の値のみが格納されることが多い．**ストレージストラテジー**は，この傾向を利用し，値のデータ型をコレクション全体の共通情報とすることで，コレクション内の値の表現を変える手法である [1]．これにより，データ型を保存するために必要であったメモリが削減される．また，小さな値のみが格納されるコレクションに対しては，1 要素に必要なメモリをさらに削減することができる [9]．

コレクションに共通のデータ型を，**ストラテジー**と呼ぶ．コレクションが持つストラテジーは，コンテナが空で要素が存在しないことを表す EmptyStrategy で始まり，コレクションに追加された値に応じて自動的に遷移する．様々な値が追加された結果，最終的に特定のデータ型に特化したストラテジーが存在しなくなった場合は，ストレージストラテジーを使わない場合と同様の方法で要素を表現する JSValueStrategy に遷移する．

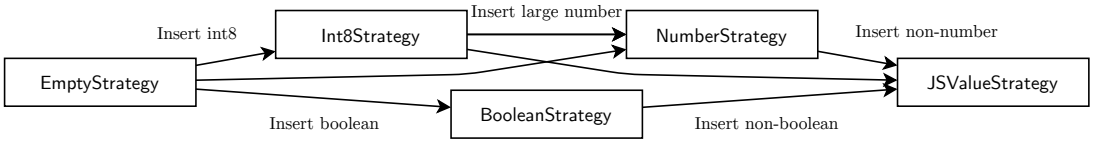


**図 4　ストレージストラテジーの遷移**

```
var array1 = [
  { p1: 123, p2: true },
  { p1: 234, p2: false },
  { p1: 345, p2: true }
];
```

**図 5　オブジェクトの配列**

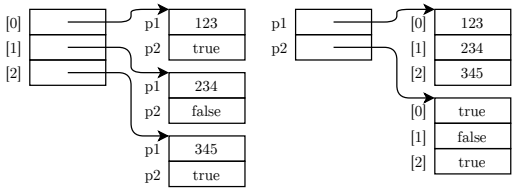


**図 6　2 つの配列表現 (左: 通常, 右: SoA 形式)**

例として，図 4 のような，8 ビット整数を表す Int8Strategy や，64 ビット浮動小数点数を表す NumberStrategy，真偽値を表す BooleanStrategy を持つストレージストラテジーにおける，図 3 のプログラムの動作を考える．まず，EmptyStrategy であった配列に 2 行目で数値 12 が追加されると，これは 8 ビット以内の整数なので，Int8Strategy に遷移する．続いて 1.23 が追加されると，8 ビット整数の範囲を超えるため，NumberStrategy に遷移する．3 行目で数値でない値が追加されると，JSValueStrategy に遷移する．

## 3　設計

ストレージストラテジーを用いてオブジェクトの配列の使用するメモリを削減する．

### 3.1　SoA 形式の配列

JavaScript の VM におけるオブジェクトの配列は，通常，オブジェクトへのポインタの配列として表される．例えば，図 6 左は，図 5 のようなプログラムにより生成される配列のメモリレイアウトである．プロパティ p1 に整数を，p2 に真偽値を持つオブジェクトへのポインタを，要素順に並べて保持している．

提案手法では，図 6 右のように，各オブジェクトの同じプロパティからなる配列を作成する．すると，p1 の配列は整数の配列，p2 の配列は真偽値の配列になり，プリミティブ型の値の配列として，ストレージストラテジーにより使用するメモリを削減できる．以後，このようなメモリレイアウトを，**SoA (Struct of Array) 形式**と呼ぶことにする．また，以降でどの配列を指すか紛らわしい場合は，SoA 形式になった配列全体を**ルート配列**，プロパティの値の配列を**プロパティ値配列**と呼ぶことにする．

### 3.2　提案手法を適用する配列の条件

提案手法による最適化を適用できるのは，配列が次の 2 つの条件を満たす場合である．

- **条件 1**: 配列の全ての要素が同一のプロパティの集合を持つオブジェクトであること．
- **条件 2**: 配列の全ての要素が他の SoA 形式の配列と共有されていないこと．

条件 1 は，オブジェクトが持つ **Hidden Class** [2] の一致により確認できる．Hidden Class とは，プロパティ名とプロパティの格納場所の対応関係を記録するデータであり，同じプロパティの集合を持つオブジェクト間で共有される．

条件 2 が必要なのは，SoA 形式の配列では，オブジェクトが配列に埋め込まれ，独立した存在ではなくなるためである．このため，一度オブジェクトが SoA 形式の配列に格納されると，そのオブジェクトが別の SoA 形式の配列の要素となることはない．なお，SoA 形式の配列以外からの参照は，Proxy オブジェクトを経由して行う．Proxy オブジェクトについては，3.5 節で詳しく述べる．

### 3.3 SoA ストラテジー

提案手法において，配列を SoA 形式により表現する状態は，ストレージストラテジーにおけるストラテジーの一種である．このストラテジーを SoAStrategy と呼ぶことにする．図 7 のように，SoAStrategy には EmptyStrategy からのみ遷移し，SoAStrategy からは JSValueStrategy にのみ遷移する．ただし，SoAStrategy はオブジェクトが持つ Hidden Class ごとに別のストラテジーと考え，異なる Hidden Class のストラテジーには遷移しない．

配列が SoAStrategy であるとき，プロパティ値配列ごとにストレージストラテジーを設定する．このストラテジーは，図 8 のように，ルート配列にオブジェクトが追加されるか，既存のオブジェクトのプロパティが変更され，あるプロパティ値配列に現在のストラテジーに収まらない値が格納されるとき，プロパティごとに独立して遷移する．

例えば，図 9 のプログラムでは，まず 3 行目で配列は EmptyStrategy から SoAStrategy に遷移する．このとき，各オブジェクトのプロパティは，NumberStrategy と BooleanStrategy に設定される．次に，5 行目で配列にオブジェクトが追加されると，NumberStrategy では表現できない文字列が設定されたプロパティ p1 のストラテジーのみが JSValueStrategy に遷移する．

### 3.4 SoA 形式の解除

次のいずれかの場合，配列は SoAStrategy で表現できなくなる．このとき，配列の SoA 形式による表現は解除され，JSValueStrategy に遷移する．

- **条件 1**: 配列にオブジェクト以外の値が追加されたとき．
- **条件 2**: 配列に異なる Hidden Class を持つオブジェクトが追加されたとき．
- **条件 3**: 配列に含まれるオブジェクトの Hidden Class が変化したとき．

図 10 は，配列の SoAStrategy が解除されるプログラムの例である．1 行目で生成された配列は，2 行目から 4 行目の操作により，それぞれ条件 1 から条件 3 を満たす．これにより，SoAStrategy は解除され，JSValueStrategy に遷移する．このとき，埋め込まれていたオブジェクトは，独立したオブジェクトに変換される．

### 3.5 Proxy オブジェクト

#### 3.5.1 配列内のオブジェクトへの参照

3.2 節で述べたように，SoA 形式の配列内に埋め込まれたオブジェクトは，ヒープ上の独立した存在ではなくなる．このため，埋め込まれたオブジェクトへの参照を保持するための仕組みが必要となる．

例えば，図 5 のプログラムで生成された配列について，式 `array1[1]` は配列 `array1` の 1 番目の要素を表す．これはオブジェクトであり，通常，内部的にはヒープ上のオブジェクトへのポインタとして表現される．しかし，提案手法では，このオブジェクトは配列内に埋め込まれているため，オブジェクトのポインタでは表現できない．

そこで，埋め込まれたオブジェクトへのポインタの代わりに，埋め込まれたオブジェクトを間接的に表す **Proxy オブジェクト**を作成して，その Proxy オブジェクトへのポインタを利用する．Proxy オブジェクトは，オブジェクトが埋め込まれた配列へのポインタと，配列内でのオブジェクトのインデックスを保持している．この Proxy オブジェクトを元のオブジェクトを同等のものとして扱えるようにすることで，利用者は透過的に埋め込まれているオブジェクトを利用できる．図 11 は，図 5 のプログラムで生成された配列の 1 番目の要素を参照する Proxy オブジェクトを作成した例である．この例では，Proxy オブジェクトはオブジェクトが埋め込まれた SoA 形式の配列へのポインタと，埋め込まれたインデックスである 1 を保持している．

オブジェクトが SoA 形式の配列に埋め込まれるときは，そのオブジェクトの領域を上書きして Proxy オブジェクトを作る．これは，そのオブジェクトが複数の箇所から参照されている可能性があるためである．

#### 3.5.2 独立のオブジェクトを指す Proxy オブジェクト

3.4 節で述べたように，配列の SoA 形式が解除されるときは，配列内のオブジェクトは独立したオブ

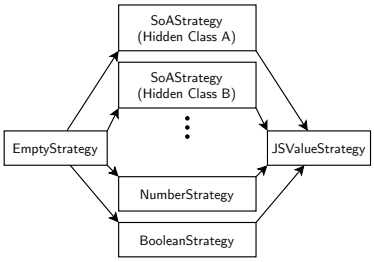


**図 7　SoAStrategy への遷移と解除**

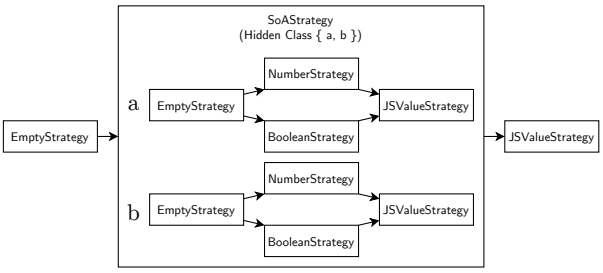


**図 8　プロパティのストラテジーの遷移**

```
1  var array1 = []; // EmptyStrategy
2  // p1: NumberStrategy, p2: BooleanStrategy
3  array1.push({ p1: 123, p2: true });
4  // p1: JSValueStrategy, p2: BooleanStrategy
5  array1.push({ p1: "text", p2: false });
```

**図 9　SoAStrategy に遷移するプログラムの例**

```
1  var array1 = [{ p1: 123, p2: true }];
2  if (...) array1.push(456); // 条件1
3  if (...) array1.push({ p3: "text" }); // 条件2
4  if (...) array1[0].p3 = "text"; // 条件3
```

**図 10　SoAStrategy が解除されるプログラムの例**

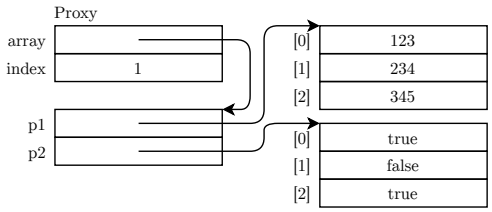


**図 11　Proxy オブジェクト**

ジェクトに変換される．また，SoA 形式の配列の要素に他のオブジェクトが代入されたときも，元の埋め込まれていたオブジェクトは独立したオブジェクトに変換される．これらの場合に，変換されていたオブジェクトを指していた Proxy オブジェクトが，同じオブジェクトを指し続けることができるように，Proxy オブジェクトを独立したオブジェクトを指す形式に変換する．これを実現するために，各配列は，その配列の各要素を指す Proxy オブジェクトを覚えておく．さらに，これを利用して，同じ要素の Proxy オブジェクトは，2 つ以上作らないようにする．

また，Proxy オブジェクトで指される独立したオブジェクトは，Proxy オブジェクト以外から直接参照されないようにする．これは，Proxy オブジェクト $P$ から指されたオブジェクト $O$ が，直接の参照を使って SoA 形式の配列に埋め込まれるのを防ぐためである．もし，直接の参照を使って SoA 形式の配列に埋め込まれると，そのオブジェクト $O$ が新しい Proxy オブジェクト $Q$ で上書きされ，$O$ を指していた Proxy オブジェクト $P$ は Proxy オブジェクト $Q$ を指すことになってしまう．このような多段の Proxy オブジェクトが作られると，メモリとリードバリアのオーバヘッドが大きくなる．

## 4　実装

提案手法を，組み込みシステム向けに開発された JavaScript VM である eJSVM [7] に実装した．eJSVM には，2 章で述べたプリミティブ型の配列に対するストレージストラテジーや，3.2 節で述べた Hidden Class による最適化が実装されている．

### 4.1　eJSVM における配列の表現

図 12 は，eJSVM における，オブジェクトの表現を示したものである．eJSVM のオブジェクトは，`JSObject` と呼ばれ，1 ワード目は常に Hidden Class へのポインタとなっている．Hidden Class は，オブジェクトが持つプロパティ名と，そのプロパティが格納されるオフセットの対応関係を記録する．

配列は，`JSObject` の一種として表現されるが，予

表 1　プリミティブ型の値のストラテジー

| 名称 | 説明 | 空要素の表現 |
|---|---|---|
| EmptyStrategy | まだ要素が追加されていない配列. | - |
| Int8Strategy | $[-2^7+1, 2^7-1]$ の範囲の整数．1 要素に 8 ビット使用する． | $-2^7$ |
| Int16Strategy | $[-2^{15}+1, 2^{15}-1]$ の範囲の整数．1 要素に 16 ビット使用する． | $-2^{15}$ |
| BooleanStrategy | 真偽値．1 要素に 2 ビット使用する．00 が `false`，01 が `true` に対応する． | 10 |
| JSValueStrategy | 任意の値. | `JS_EMPTY` (定数) |

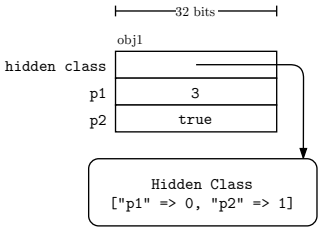


図 12　eJSVM におけるオブジェクトの表現

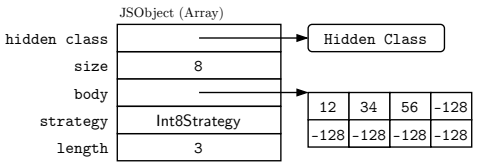


図 13　eJSVM における配列の表現

め定義された `size`，`body`，`strategy`，`length` というプロパティが自動的に作成される．例えば，図 13 は，配列 `[12, 34, 56]` の eJSVM 上での表現の例である．この配列には小さな整数しか格納されていないため，`strategy` プロパティは Int8Strategy になる．また，要素数は 3 であるため，`length` プロパティは 3 だが，内部的には 8 個分の領域が確保されているため，`size` プロパティは 8 となっている．`body` は，配列の要素を格納する領域へのポインタである．

### 4.2　SoAStrategy の導入

提案手法の実装で，図 5 のような配列が SoAStrategy によって表される場合のメモリレイアウトを図 14 に示す．配列が SoAStrategy を持つとき，配列の `strategy` の値は，配列内のオブジェクトの Hidden Class へのポインタとなる．また，`body` の値は，配列内のオブジェクトのプロパティ一つにつき 2 ワードと，追加で 1 ワード分の領域を持つ `SoAArrayBody` へのポインタとなる．

`SoAArrayBody` の最初のワードは，配列の各要素のメタデータを格納する `ElementAttributes` へのポインタである．`ElementAttributes` 領域についての詳細は，4.3 節で述べる．2 ワード目以降は，各プロパティごとのストラテジーと，実際のプロパティの値が保存されている領域へのポインタの繰り返しである．

### 4.3　Proxy と要素のメタデータ

`ElementAttributes` は，配列の 1 要素につき 1 ワード分の領域を持つ．この値は，配列内の対応する要素の状態に応じて，次の 3 つの値のいずれかをとる．

- Proxy オブジェクトへのポインタ：要素が存在し，Proxy オブジェクトが存在する場合
- `JS_NON_EMPTY` (定数)：要素が存在するが，Proxy オブジェクトが存在しない場合
- `JS_EMPTY` (定数)：要素が存在しない場合

`ElementAttributes` 領域が持つ Proxy オブジェクトへのポインタは弱参照になっており，Proxy オブジェクトが他から参照されていないときは，ガーベージコレクション (GC) によって回収される．

## 5　評価

4 章で示した実装を用い，提案手法実装前の eJSVM をベースラインとしてメモリ使用量の変化を調査した．加えて，提案手法の実装によるオーバーヘッドの大きさを確認するため，実行時間も計測した．

### 5.1　実験設定

メモリの使用量は，プログラムが 3 分以内に正常に終了する最小のヒープサイズとして測定した．GC を実装した処理系では，ヒープサイズが理論上の限界

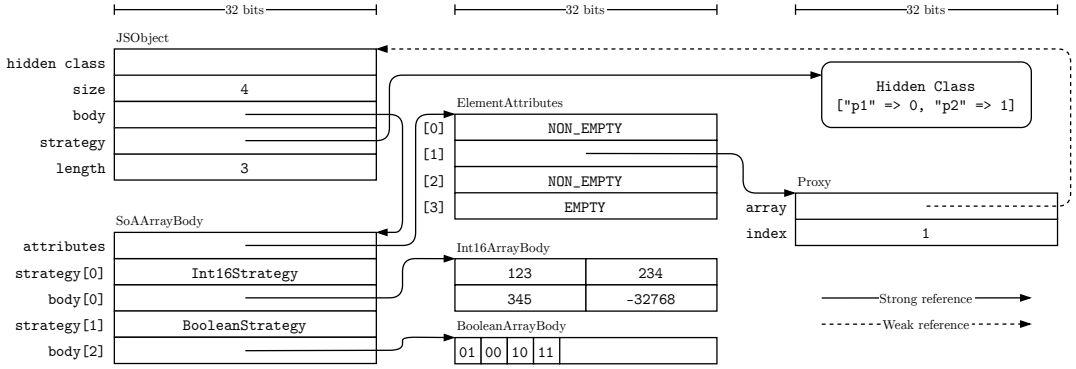


**図 14　提案手法実装後のメモリレイアウト**

**表 2　ベンチマークプログラム**

| ベンチマークプログラム集 | プログラム |
|---|---|
| センサの値を蓄積して送信するプログラム (図 2) | Sensor |
| Are we fast yet Benchmark [5] | Bounce, CD, DeltaBlue, Havlak, Json, List, Mandelbrot, NBody, Permute, Queens, Richards, Sieve, Storage, Towers |
| eJSVM 独自のベンチマークプログラム [8] | dht11 |
| 組み込みシステム向けベンチマークプログラム [6] | CRC32, IIR_BQ_F32, FIR_F32, FFT_I16, SHA256 |

に近づくにつれ，GC の頻度が急激に上昇する．ベンチマークプログラムは，ヒープサイズが十分大きいときには 10 秒程度で実行が終了するように調整されており，実行時間が 3 分を超えるヒープサイズは，理論上の限界に十分近いと考えられるためである．一方，実行時間については，ヒープサイズをプログラムが必要とするメモリに対して十分大きい，128 MB に設定して計測した．

ベンチマークには，図 2 で示したセンサの値を受け取って送信するプログラム Sensor を用いた．引数として，`temperature` 引数には 0 以上 100 未満の，`humidity` 引数には 0 以上 1000 未満のランダムな整数を指定した．

また，一般的なプログラムにおける提案手法のオーバーヘッドを確認するため，表 2 に示した 4 つのベンチマークプログラム集を用いた．このうち，dht11[†1] は温度センサーから送られるデータのデコードプログラムで，CRC32, IIR_BQ_F32, FIR_F32, FFT_I16, SHA256 は組み込みシステムを想定したベンチマークプログラム集 [6] を JavaScript に移植したもの [†1] である．

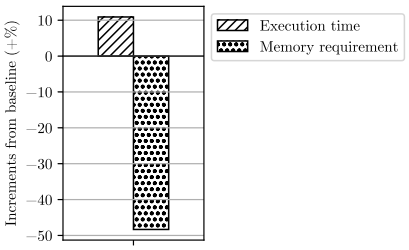


**図 15　Sensor の実行時間とメモリ使用量**

計算機の CPU には Intel Xeon W-2235 (クロックは 3.80 GHz に固定) を用い，OS には Linux 5.15 Ubuntu 20.04.4 を用いた．

### 5.2　メモリ使用量と実行時間

図 15 は，提案手法を実装した VM における，Sensor のメモリ使用量と実行時間を，提案手法実装前と

†1　eJSBenchmarks:
https://github.com/plasgroup/eJSBenchmarks

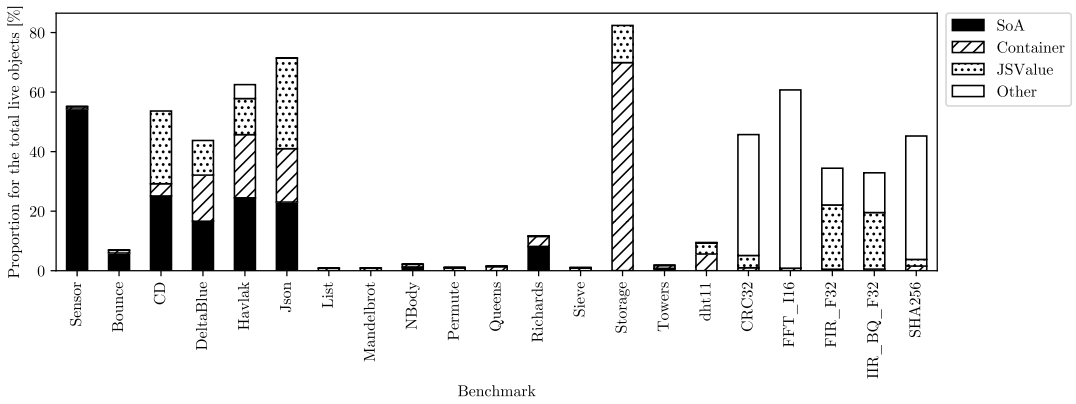


**図 16　ストラテジーの使用状況**

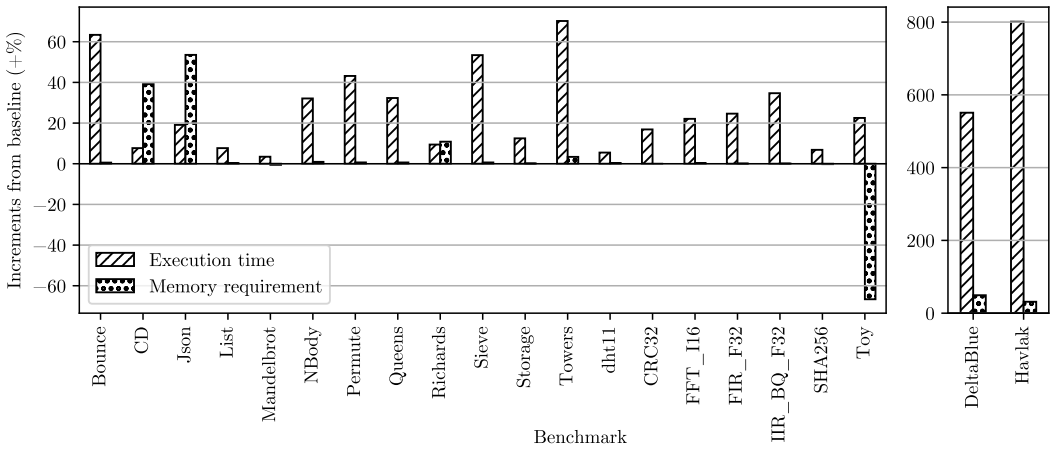


**図 17　提案手法の実行時間とメモリ使用量**

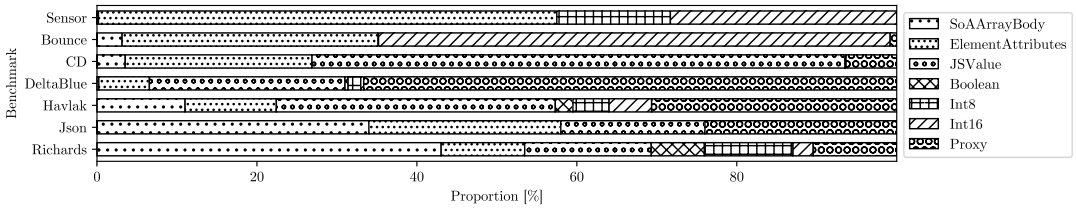


**図 18　プロパティごとのストラテジーの使用状況**

比較した図である．Sensor のメモリ使用量は約 50%減少し，実行時間は約 10%増加した．

図 16 は，ヒープ上に生存している全てのオブジェクトの中で，各ストラテジーを持つ配列が占める割合を示したものである．Container はストラテジーによらず共通する JSObject の領域で，それ以外は各ストラテジーが個別に管理する領域である．ヒープサイズを十分大きく設定したうえで，ガベージコレクタを一

定数のバイトコード命令が実行されるたびに起動し，生き残ったオブジェクトを数え上げて平均をとった．その結果，Sensor では，生きているオブジェクトが占めるメモリの約 50%が SoAStrategy を持つ配列であることが分かった．

さらに，図 18 は，図 16 で SoAStrategy として分類された領域の内訳を示している．`SoAArrayBody`，`ElementAttributes`，`Proxy` は各プロパティが持つストラテジーによらず共通する領域で，それ以外は配列内のオブジェクトのプロパティが保存される領域である．Sensor は，0 以上 100 未満の整数と，0 以上 1000 未満の整数の 2 つのプロパティを持つオブジェクトを配列に格納している．このため，Int8Strategy と `Int16Strategy` が大きな割合を占めている．eJSVM では通常一つの値を 32 ビットで表すため，Int8Strategy ではサイズが 4 分の 1 に，Int16Strategy ではサイズが 2 分の 1 になる．これが，図 15 におけるメモリ使用量の減少につながったと考えられる．

### 5.3 汎用的なベンチマークプログラムでの結果

図 17 は，Sensor 以外のベンチマークプログラムについて，提案手法実装前後のメモリ使用量と実行時間を比較したものである．図 16 より，SoAStrategy が使用されていた主なベンチマークプログラムは Sensor, Bounce, CD, DeltaBlue, Havlak, Json, Richards であったため，これらのベンチマークプログラムの結果について考察する．

まず，メモリ使用量については，多くのベンチマークプログラムではほとんど変化していないが，CD, DeltaBlue, Havlak, Json, Richards では，最大で 60% 程度増加している．これは，SoAStrategy の使用による空間的なオーバーヘッドが，メモリ使用量の増加につながったためだと考えられる．

図 18 は，各ベンチマークプログラムでメモリ使用量が増加した原因を示している．CD や Json では，メモリ使用量を大きく減らすことのできる BooleanStrategy, Int8Strategy, Int16Strategy が使用されていない．また，DeltaBlue や Havlak，Json では，SoAStrategy がなければ必要のない Proxy オブジェクトが多く作られている．Richards については Int8Strategy と Int16Strategy，BooleanStrategy のいずれも使用されているが，`SoAArrayBody` に対してその割合は小さいため，要素数が少ない配列が多く作られていると考えられる．このため，全体としてはメモリ使用量は増加した．Bounce のメモリ使用量は，提案手法実装前後でほとんど変化していない．これは，SoAStrategy を使用している配列の割合が非常に小さいためである．

実行時間については，全てのベンチマークで増加した．増加幅はベンチマークプログラムによって異なるが，DeltaBlue と Havlak は突出して多く，5 倍から 8 倍の実行時間を要している．これは，Proxy オブジェクトの作成や，一度埋め込まれたオブジェクトの取り出しのためにメモリアロケーションが多く発生していることや，Proxy オブジェクトを透過的に扱えるようにするためのリードバリアやライトバリアのオーバーヘッドが大きかったことが原因だと考えられる．

## 6 関連研究

Bolz ら [1] は，ストレージストラテジーを用いることで，Python の VM のコレクション操作が高速化されることを示した．また，同様の手法は，JavaScript の VM においても限定的に実装されている [2]．

Makor らは，オブジェクトの配列を自動的にプロパティごとの配列に変換する手法を提案した [4]．また，ネストされたオブジェクトも同様に扱う手法を示した [3]．本研究は，変換後のプロパティごとの配列に対して，さらにストレージストラテジーを適用しているという点で異なっている．

## 7 おわりに

本研究では，JavaScript の配列におけるメモリ使用量の削減を目的として，オブジェクトの配列をプロパティごとに分けてストレージストラテジーを適用する手法を提案した．これにより，組み込みシステム向けのプログラムにおいて，センサのデータをオブジェクトとして配列に保存しておくというユースケースにおいて，提案手法が有効であることを示した．一方，その他のベンチマークプログラムでは，メモリ使用量が増加した．これは，メモリ使用量の削減ができ

ない配列に対しても，提案手法を適用してしまっているためである．今後の課題には，提案手法を効果のある配列のみに適用することで，オーバーヘッドを減らすことが挙げられる．

**謝辞** 本研究の一部は，JSPS 科研費 18KK0315 の助成を受けたものです．

## 参 考 文 献

[1] Bolz, C. F., Diekmann, L., and Tratt, L.: Storage strategies for collections in dynamically typed languages, *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, (2013), pp. 167–182.

[2] Hosking, A. L., Bond, M., Clifford, D., Payer, H., Stanton, M., and Titzer, B. L.: Memento mori: dynamic allocation-site-based optimizations, *Proceedings of the 2015 International Symposium on Memory Management*, (2015), pp. 105–117.

[3] Kloibhofer, S., Makor, L., Leopoldseder, D., Bonetta, D., Stadler, L., and Mössenböck, H.: Control Flow Duplication for Columnar Arrays in a Dynamic Compiler, *The Art, Science, and Engineering of Programming*, Vol. 7, No. 3(2023).

[4] Makor, L., Kloibhofer, S., Leopoldseder, D., Bonetta, D., Stadler, L., and Mössenböck, H.: Automatic Array Transformation to Columnar Storage at Run Time, *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, (2022), pp. 16–28.

[5] Marr, S., Daloze, B., and Mössenböck, H.: Cross-language compiler benchmarking: are we fast yet?, *Proceedings of the 12th Symposium on Dynamic Languages*, (2016), pp. 120–131.

[6] Plauska, I., Liutkevičius, A., and Janavičiūtė, A.: Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller, *Electronics*, Vol. 12, No. 1(2022), pp. 143.

[7] Ugawa, T., Iwasaki, H., and Kataoka, T.: eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems, *Journal of Computer Languages*, Vol. 51(2019), pp. 261–279.

[8] 小野澤拓, 岩崎英哉, 鵜川始陽: アプリケーションと実行環境に適応したカスタマイズが可能な JavaScript 処理系, コンピュータ ソフトウェア, Vol. 38, No. 3(2021), pp. 3_23–3_40.

[9] 永谷龍彦, 鵜川始陽: ストレージストラテジーによる組み込み向け JavaScript バーチャルマシンのメモリ使用量の削減, 日本ソフトウェア科学会第 *39* 回大会講演論文集, August 2022.