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

永谷 龍彦　鵜川 始陽

JavaScript をはじめとする動的型付け言語では，静的型付け言語と比べ，多くのメモリが要求される．これは，動的型付け言語は実行時にデータ型を決定するため，値とともにデータ型もメモリに保存しておく必要があるためである．しかし，動的型付け言語においても，配列などのコレクションは，同じ型の値を格納するために用いられることが多い．本研究では，同じ型の値だけが格納されたコレクションに対して，データではなくコレクションにデータ型の情報を持たせるストレージストラテジーという手法を，配列に適用してメモリ使用量を削減する．さらにメモリ使用量を削減するために，配列のデータ型には 1 バイト整数のような 1 ワードより小さいデータ型を導入する．これを組み込み向けシステム向け JavaScript バーチャルマシンに実装して評価した．

1　はじめに

JavaScript などの動的型付け言語の普及は，ソフトウェア開発の敷居を大幅に引き下げたが，組み込みシステム向けの開発では未だ C や C++ といった伝統的な言語が用いられる場合が多い．これは，組み込みシステム向け開発では，メモリサイズの制約が特に厳しいためである．

本研究では，ストレージストラテジー [1] を，JavaScript 仮想機械 (VM) の配列 (`Array` オブジェクト) の実装に用い，メモリ使用量を削減する．ストレージストラテジーは，リストや辞書などのコレクションに格納された値のメモリ表現を，追加された値のデータ型に応じて自動的に最適化する手法である．本研究では，値のデータ型だけでなく，その大きさに着目し，必要に応じて各要素を 1 ワードより小さな領域に配置することで，高いメモリ効率を実現する．

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，メモリ使用量を平均 13% 削減することに成功した．実行時間のオーバーヘッドは平均 3.8% であった．

2　ストレージストラテジー

動的型付け言語のメモリ使用量は静的型付け言語より多くなる．これは，プログラムに型を記述しないため，値ごとに型を記録する必要があるからである．単純な VM の実装では，全ての値についてボクシングを行う．すなわち，値の実体と型情報をメモリ上の別の場所に配置しておき，本体にはそのアドレスのみを持たせる．配列のようなコレクションにおいては，図 1 右のように全ての値に対してボクシングが行われることになり，図 1 左のような C や C++ の配列と比べて無駄が多くなる．

ストレージストラテジーが実装された VM では，図 2 のように，コレクションの値のデータ型が各要素にではなくコレクション本体に格納されるようになる．この設計は，動的型付け言語であっても，コレクションには同じデータ型の値が格納される場合が多い [1] ことに由来する．すなわち，数値や文字列，オブジェクトといった異なる種類のデータが同じコレク

Reducing Memory Footprint of JavaScript Virtual Machines for Embedded Systems 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.

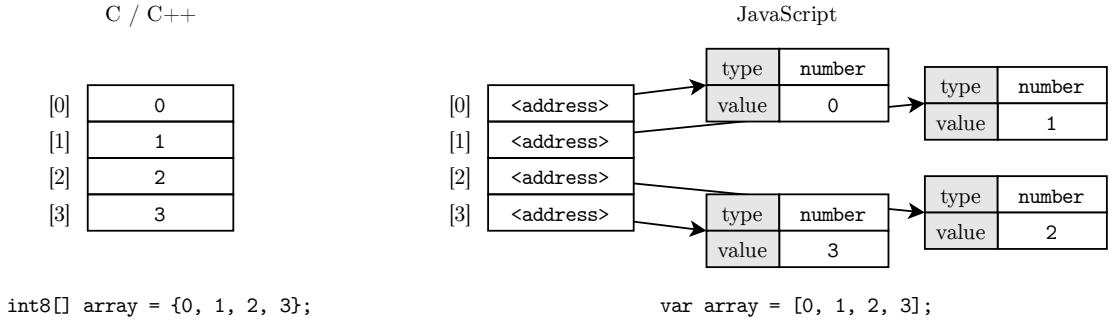

図 1　C や C++ と JavaScript の配列の表現の違い．左: C / C++, 右: JavaScript

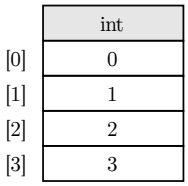

図 2　ストレージストラテジーによる配列の表現

ションに格納されていることは少ない．この変更により，コレクションのメモリ使用量が，各要素が保持していた型情報の分だけ削減される．

コレクションごとに定まるデータ型を，ストラテジーと呼ぶ．ストラテジーは，コレクションに追加された値に応じて自動的に決定される．図 3 のように，コレクションが作成された時点では，未だ何も値が格納されていないことを表す EmptyStrategy が設定され，初めて値が格納された時点でその値に特化したストラテジーに遷移する．また，ストラテジーが対応できない要素が追加された場合，より汎用的なストラテジーに遷移する．一度汎用的なストラテジーに遷移したコレクションは，コレクションが空にされない限り，特化したストラテジーに戻ることはない．

例えば，図 4 のプログラムの場合，1 行目終了時点では `list1` は EmptyStrategy だが，2 行目には整数のみを格納することができる IntegerStrategy に遷移する．3 行目では文字列がリストに追加されているが，文字列は IntegerStrategy では対応できないため，ストラテジーはより汎用的な ObjectStrategy に遷移する．ObjectStrategy は，任意の値を格納できるストラテジーで，このストラテジーを持つコレクションの値は，ストレージストラテジーを導入していない VM と同一の方法で格納される．

3　設計

ストレージストラテジーは本来様々なコレクションに対して適用可能な手法だが，本研究は JavaScript を対象とするため，JavaScript の仕様である ECMAScript 5 [2] 時点で利用できるコレクションとして，配列のみに適用する．また，実行速度の改善ではなく，メモリ使用量の削減を優先した設計を行う．

3.1　データサイズに着目したストラテジー

提案手法では，小さな値のみを格納できるストラテジーを導入することで，メモリの使用量を積極的に削減する．Bolz らの提案したストレージストラテジーは，整数や文字列など，データの種類のみに着目していた．しかし，データの種類だけでなく，その大きさについても同様の性質が存在すると考えられる．すなわち，小さな値がほとんどを占めるコレクションにおいて，少数の要素のみが大きい値となることは起

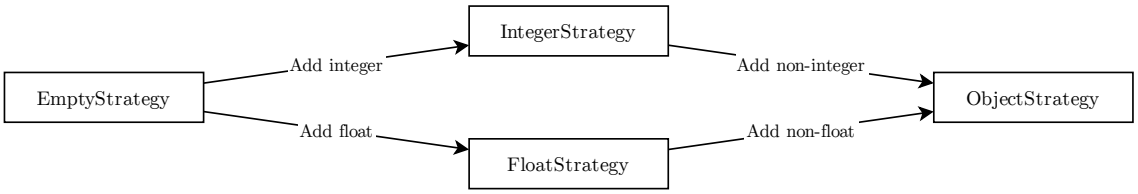

図 3　ストレージストラテジーの遷移

```
1 list1 = []
2 list1.append(123)
3 list1.append("foo")
```

図 4　ストレージストラテジーが遷移する例

こりづらい．このため，小さな値のみが格納できるストラテジーを用意することで，大きな値も格納できるよう，余分なメモリを使用する必要がなくなる．

図 5 は，数値のみが格納された配列を効率的に扱うストラテジーの組み合わせの例である．JavaScript の数値は，仕様により 64 ビット浮動小数点数と定義されているが，異なるデータ型の値が混在する配列があまり用いられないのと同様に，整数と浮動小数点数が混在する配列や，大きな整数と小さな整数が混在する配列もあまり用いられないと考えられる．このため，図 5 の例では，仕様上すべての数値を表現できる NumberStrategy のほかに，そのサブセットである 8 ビット符号付き整数のみを表現できる Int8Strategy を導入している．この VM で，図 6 に示したプログラムを実行すると，まずは 1 行目で EmptyStrategy の配列が生成される．続いて，2 行目で配列に整数 100 が追加されると，これは 8 ビット符号付き整数の範囲に収まる値なので，Int8Strategy に遷移する．さらに整数 1000 が追加されると，より汎用的な NumberStrategy に遷移し，最終的に適切なストラテジーが存在しなければ JSValueStrategy となる．

この実装により，Int8Strategy が利用されれば，NumberStrategy しか実装しない場合に比べ，メモリ使用量を 8 分の 1 に削減できる．ただし，Int8Strategy が適用された後，その範囲に収まらない数値が格納されようとした場合，NumberStrategy として必要なメモリを確保し直す必要がある．

3.2　空要素の表現

JavaScript では，図 7 のプログラムのように，`Array` コンストラクタの引数に要素数を指定して配列を作成することで，要素の一部を空にすることができる．ここでいう空とは，図 8 右のように，未定義を表す JavaScript の値である `undefined` が格納されている状態とは異なり，図 8 左のように，配列の要素自体が存在しない状態を表している．

この仕様は，JavaScript の配列がインデックスをキーとする辞書の形で定義されていることに由来しており，要素が存在しないインデックスでのアクセスでは，同名のプロパティを持つ継承元のオブジェクトを探索する必要がある．ただ，実用上は配列を敢えて辞書のように用いることはほとんどないため，VM の実装においては，配列の要素はメモリ空間に連続的に配置される．空要素を表現するには，空である状態を表現するための内部的な値を用意し，その場所に格納しておけばよい．

本研究のストレージストラテジーの実装では，空要素をどう保持するかはストラテジー側の責務とする．このため，各ストラテジーが表現できる値の範囲は，1 要素に割り当てられたメモリ領域を用いて表現可能な全ての値よりも 1 つ以上少なくなる．例えば，`true` または `false` のみを表現できる BooleanStrategy の 1 要素には， 2 bit を割り当てる．

各ストラテジーに空要素を扱う責務を持たせず，一つでも空要素を持つ配列を全て JSValueStrategy とするのは適切ではない．これは，図 9 のように，`Array` コンストラクタに要素数を指定して配列を作成した場合，初期状態で配列の各要素は空となるためである．このプログラムでは，配列には明らかに整数しか格納されないが，1 行目で `Array` コンストラクタに要素数を指定しているため，JSValueStrategy が適用

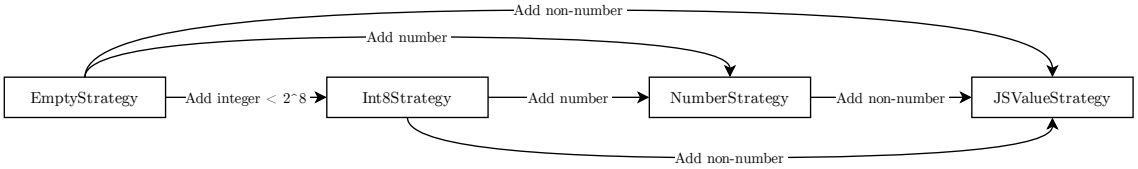

図 5　値の大きさを考慮したストラテジー遷移の例 (JSValue は JavaScript の任意の値を表す)

```
1 array = [];
2 array.push(100);
3 array.push(1000);
4 array.push("foo");
```

図 6　値の大きさを考慮してストラテジーが選択される例

```
1 array = [];
2 array[2] = "foo"; // [ <2 empty items>, 'foo' ]
3 array[0] = undefined; // [ undefined, <1 empty
    item>, 'foo' ]
```

図 7　配列の空要素が生じるプログラム

されてしまう.

3.3　EmptyStrategy とリストの長さの関係

3.2 節で提示したように，JavaScript では，`Array` コンストラクタを用いて配列の要素を入れる前に長さを指定できる．このため，配列のストラテジーが EmptyStrategy であっても，配列の長さが 0 になるとは限らない．図 9 の 1 行目のように，`Array` コンストラクタにより要素数を指定して配列が作成された場合，ストラテジーは EmptyStrategy だが，配列の長さは正となる．

これによって，配列本体のメモリを確保するタイミングが，配列を作成した時点から，配列に初めて値が追加された時点に遅延される．図 9 のプログラムの場合，1 行目では配列のコンテナのみを用意し，2 行目で初めて Int8Strategy の 3 要素分のメモリを確保される．

4　実装

提案手法を eJSVM [7] に実装した．eJSVM は，組み込みシステム向けに開発された JavaScript VM である．搭載メモリが非常に小さい環境での動作を想定し，消費メモリが少なくなるよう設計されている．

4.1　前提: eJSVM における値と配列の表現

図 10 は，eJSVM における値の表現を表している．eJSVM では，値は 64 bit で表現され，下位 3 ビットがタグとして値の種類を区別するために用いられている．残りの 61 ビットの中に収まる整数や真理値，`undefined` などの特殊な値は直接埋め込まれ，浮動小数点数やオブジェクトは，タグ付きポインタ [4] の形で表現されている [7]．この 64 ビットを `JSValue` と呼ぶ.

オブジェクトは `JSValue` からタグを外して得られるアドレスに保存されており，これを `JSObject` と呼ぶ．オブジェクトが持つプロパティは全て `JSValue` と同じ 64 bit で表現される.

Array オブジェクトは `JSObject` の特殊なケースとして扱われ，ECMAScript の定める仕様を満たすために，次のような特殊なプロパティが作成される.

- `length`: プログラムから見える配列の長さ
- `body`: 配列内部の値が `JSValue[]` の形で格納されたアドレスへのポインタ
- `size`: そのポインタが指し示す先の空間の大きさ

4.2　ストラテジーの導入

ストレージストラテジーの導入にあたり，図 11 のように，4.1 節の配列オブジェクトのスロットの部分を，次のように変更する.

- 配列の現在のストラテジーを格納する `strategy` スロットを追加する
- `body` スロットの型は `JSValue[]` ではなくストラテジーに応じて変わる

ストラテジーは，配列に要素が保存される際に，受

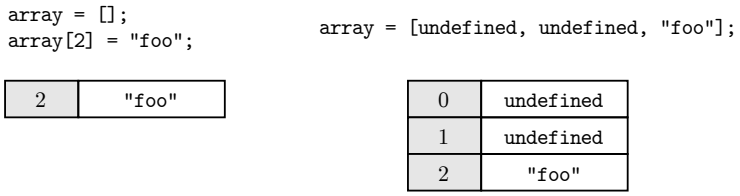

図 8　左: 空要素を含む配列, 右: `undefined` が入った配列

```
1 array = Array(3);
2 array[0] = 12;
3 array[1] = 34;
4 array[2] = 56;
```

図 9　JSValueStrategy が誤って適用される例

け取った JSValue をそれぞれに最適化された表現に変換する．また，配列から要素が取り出される際は，再度 JSValue の形に戻す．

このインターフェースに沿って，次に挙げた 6 つのストラテジーを実装する．

- EmptyStrategy: まだ要素が追加されていない配列のストラテジー．
- Int8Strategy: $[-2^7+1, 2^7-1]$ の範囲の整数を 8 ビット符号付き整数で保存するストラテジー．-2^7 は空要素を表す値とする．
- Int16Strategy: $[-2^{15}+1, 2^{15}-1]$ の範囲の整数を 16 ビット符号付き整数で保存するストラテジー．-2^{15} は空要素を表す値とする．
- Float64Strategy: 64 ビット浮動小数点数を保存するストラテジー．NaN は空要素を表す値とする．
- BooleanStrategy: 2 ビットで真理値を保存するストラテジー．00 が `false`，01 が `true`，10 が空要素を表すとする．
- JSValueStrategy: JSValue を保存するストラテジー．ストレージストラテジー導入前の実装と同等．

4.3　ストラテジーの遷移

ストラテジー間の遷移は，図 12 に示された条件で行われる．数値を表す Int8Strategy, Int16Strategy, Float64Strategy の間で遷移する場合は，最もサイズの小さいものが自動的に選択される．

ストラテジーが遷移する際，既に保存されている要素は新しいストラテジーに合わせて変換しなければならないが，最適化のため，一度 JSValue を経由することなく，専用のコードパスによって変換する．

4.4　浮動小数点数のボクシング

Float64Strategy の導入により，配列からの要素の取得は，メモリアロケーションが発生する操作となる．4.1 節で提示したように，eJSVM ではタグ付きポインタを用い，JSValue のタグを除いた領域に埋め込める値は直接埋め込んでいる．しかし，浮動小数点数は 64 bit 全てを消費するため，ポインタの形で JSValue に格納されている．Float64Strategy は浮動小数点数を値の格納の際にアンボクシングして保存するため，値の取り出しの際には再びボクシングする必要があるためである．

5　評価

第 4 章で示した実装を用いて，メモリ使用量の削減性能を評価した．各ストラテジーは EmptyStrategy と JSValueStrategy を除いて個別に無効化できるできるようにし，それぞれの結果に対する貢献を調べた．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．さらに，それぞれのストラテジーが，各ベンチマークプログラムどれだけ使用されているかを調査した．

5.1　実験設定

ベンチマークプログラムには，Marr らの Cross-language compiler benchmarking [6] と，独自に作成した温度センサーの計測データの解析プログラム

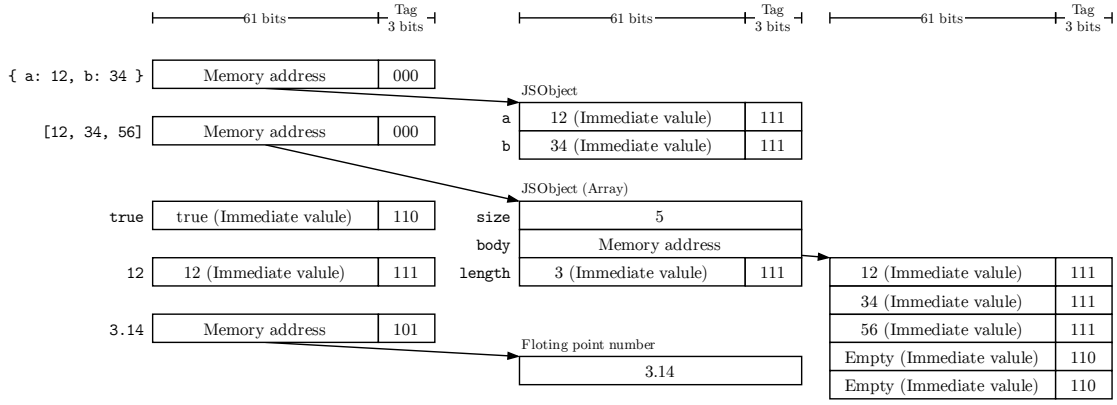

図 10　eJSVM における値の表現

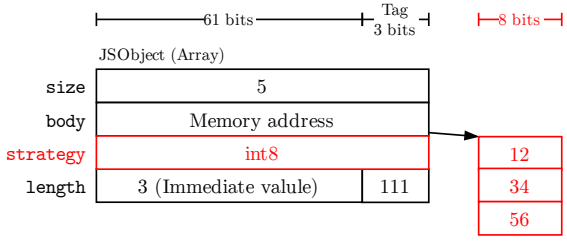

図 11　提案手法導入後の配列表現

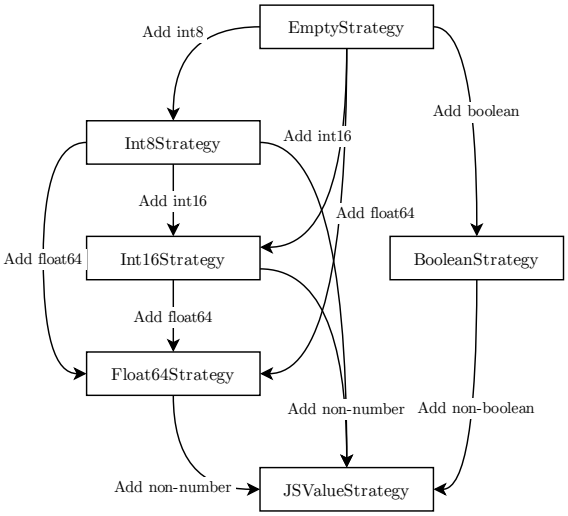

図 12　実装したストラテジーの遷移

dht11 を用いた．

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．精度は 128 KB である．これは，ガベージコレクタを実装した処理系でヒープサイズを小さくすると，ガベージコレクションの頻度が急激に上昇し，実行時間が伸びてしまうためである．

実行時間は，ヒープサイズを 128 MB に設定した

うえで計測した．128 MB はこのベンチマークプログラムに対して十分な大きさで，これ以上ヒープサイズを増やしても実行時間はほとんど変わらないことを確認している．

比較対象は，次の 9 種類である．

- 提案手法実装前 (baseline)
- EmptyStrategy と JSValueStrategy のみ実装したもの (none)
- 加えて Int8Strategy を実装したもの (int8)
- 加えて Int16Strategy を実装したもの (int8,int16)
- 加えて Float64Strategy を実装したもの (int8,int16,float64)
- 前 4 つの構成に BooleanStrategy を加えたもの (boolean[,int8[,int16[,float64]]])

ストラテジーの使用状況の調査では，全てのストラテジーを有効にした上で，新しく配列が作成されるときと，配列のストラテジーが遷移するときにその遷移元と遷移先を記録することで，各ストラテジーに最終的に到達した `Array` オブジェクトを数えた．また，`Array` オブジェクトがプログラム全体で必要なメモリな中でどれだけ支配的なのかを調べるため，ガベージコレクションが発生した際に，確保されたオブジェクト全体と生存しているオブジェクト全体のメモリ使用量の中で `Array` オブジェクトが占める割合を計測した．

実験環境の計算機の CPU は Intel Xeon W-2235 (6 コア，6 スレッド，クロックは 3.80 GHz に固定) を，DRAM は 32 GB の DDR4 を搭載した．OS は Linux 5.15 Ubuntu 20.04.4 を用いた．

5.2 メモリの使用量と実行時間

図 13 は，メモリの使用量と実行時間の baseline 比での増加率の，全ベンチマークでの幾何平均である．none と int8，none と int8 の比較により，Int8Strategy と BooleanStrategy によるメモリ使用量の減少率が大きいことが分かる．また，EmptyStrategy のみを実装した none もメモリ使用量を 3% 程度減少させている．一方，Int16Strategy や Float64Strategy の効果は限定的である．実行時間については，特に Float64Strategy と BooleanStrategy の導入による影響が大きい．

この結果をベンチマークプログラムごとに表したのが図 14 である．著しい変化を示しているのが Havlak, Sieve, Storage, dht11 の 4 つで，25% 〜 50% 程度メモリ使用量が減少している．その中でも，Havlak と Storage に関しては，EmptyStrategy と JSValueStrategy しか実装されていないにも関わらず，メモリ使用量は大幅に減少している．一方，Sieve は BooleanStrategy が実装されているとき，`dht11` は Int8Strategy が実装されているときにメモリ使用量の減少が起きている．Havlak については Int16Strategy による効果も見て取れる．

すべてのベンチマークプログラムの平均からは，実装するストラテジーの数が増えれば増えるほど，メモリの使用量は減少するが，実行時間が若干増加していることが読み取れる．

5.3 ストラテジーの使用状況

図 15 は，各ベンチマークプログラムにおいてそれぞれのストラテジーに最終的に到達した `Array` オブジェクトの数の比率を表している．また，図 16 は，確保されたオブジェクト全体 (Allocated) のサイズと，ガベージコレクションが起きた時点で生存しているオブジェクト全体 (Live) のサイズのうち，それぞれ `Array` オブジェクトが占める割合を表している．Havlak, Json, List, Mandelbrot, NBody, Permute, Richards, Storage は，全て EmptyStrategy で終わる配列が多い．このうち，生存オブジェクトの多くを `Array` オブジェクトが多くを占めている Havlak と Storage のメモリ使用量が大幅に削減されていることから，これらのメモリ使用量の減少の理由が，初期化されたまま値の追加が行われない配列の存在にあると結論付けられる．

Queens や Sieve は，BooleanStrategy が多く使用されている．Sieve に関しては生存オブジェクトに占める Array オブジェクトの割合は高くないものの，確保されたメモリ全体に占める配列の割合は高く，また，Sieve のアルゴリズムが内部的に非常に長い Array オブジェクトを用いていることから，提案手法によりメモリ使用量が減少したものと考えられる．Queens は

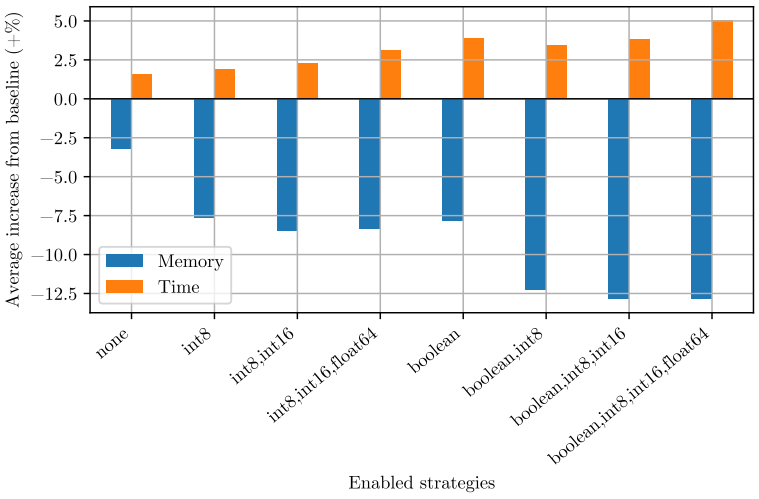

図 13　全ベンチマークプログラムの平均の消費メモリと実行時間（対ベースライン）

Benchmark	Strategy	none	int8	int8 int16	int8 int16 float64	boolean	boolean int8	boolean int8 int16	boolean int8 int16 float64
Bounce	memory (+% to baseline)	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16
	time (+% to baseline)	4.92	1.82	3.79	0.94	1.45	4.79	0.04	2.35
CD	memory (+% to baseline)	0.94	0.92	0.94	0.92	0.92	0.94	0.92	0.92
	time (+% to baseline)	-0.32	0.04	0.27	-0.13	0.10	-0.02	-0.39	1.98
DeltaBlue	memory (+% to baseline)	3.36	3.35	3.35	3.35	3.35	3.35	3.35	3.35
	time (+% to baseline)	2.16	2.58	4.16	3.55	2.73	4.25	2.33	3.75
Havlak	memory (+% to baseline)	-17.35	-17.63	-26.96	-26.96	-17.35	-17.63	-26.97	-26.97
	time (+% to baseline)	-1.24	-1.83	-1.56	-0.94	-0.61	-0.32	-1.33	0.45
Json	memory (+% to baseline)	1.18	-4.14	-4.13	-4.14	1.18	-4.13	-4.14	-4.14
	time (+% to baseline)	0.01	1.23	0.43	1.43	1.96	2.50	1.12	2.82
List	memory (+% to baseline)	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18
	time (+% to baseline)	6.32	4.70	5.64	10.35	8.77	9.11	12.80	12.28
Mandelbrot	memory (+% to baseline)	0.00	0.97	0.16	0.81	0.81	0.00	0.81	0.97
	time (+% to baseline)	-4.73	-0.49	-0.69	2.65	5.46	-5.71	3.12	1.89
Nbody	memory (+% to baseline)	0.14	0.99	0.14	0.99	0.99	0.28	0.99	0.99
	time (+% to baseline)	2.99	1.65	4.32	4.90	5.57	2.83	5.06	4.10
Permute	memory (+% to baseline)	0.19	0.00	0.00	0.00	0.19	0.00	0.00	0.00
	time (+% to baseline)	6.35	6.20	9.42	7.04	5.63	10.17	6.19	9.18
Queens	memory (+% to baseline)	0.00	0.00	0.00	0.00	-0.54	-0.71	-0.71	-0.71
	time (+% to baseline)	2.67	3.40	3.29	5.39	6.07	7.82	7.06	9.74
Richards	memory (+% to baseline)	-0.25	-0.25	-0.25	-0.25	-0.25	-0.25	-0.25	-0.25
	time (+% to baseline)	1.75	0.96	1.09	-0.90	-2.85	3.77	-1.84	0.96
Sieve	memory (+% to baseline)	0.09	0.09	0.09	0.09	-52.53	-52.53	-52.53	-52.53
	time (+% to baseline)	-0.74	7.90	2.94	7.72	14.39	12.83	14.86	18.33
Storage	memory (+% to baseline)	-30.39	-30.40	-30.39	-30.42	-30.42	-30.39	-30.42	-30.42
	time (+% to baseline)	-3.28	-3.53	-3.88	-3.29	-3.40	-4.26	-3.52	-1.22
Towers	memory (+% to baseline)	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18
	time (+% to baseline)	7.37	4.89	7.39	6.38	6.38	9.42	10.31	6.40
dht11	memory (+% to baseline)	0.56	-47.97	-47.97	-47.97	0.45	-47.97	-47.97	-47.97
	time (+% to baseline)	0.56	-0.19	-1.76	2.75	7.90	-3.28	3.00	4.03

図 14　実験結果の内訳

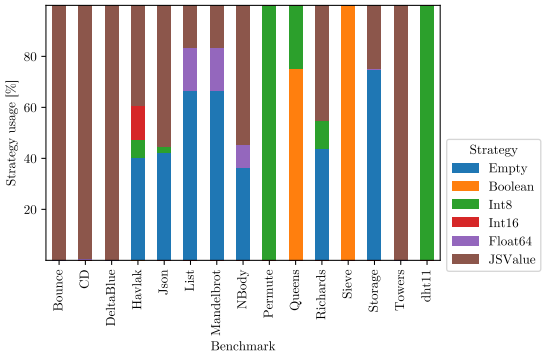

図 15　ストラテジーの使用状況

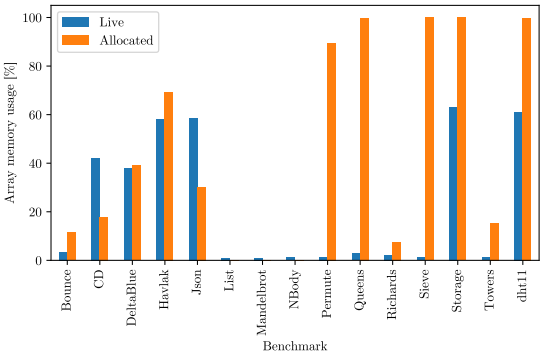

図 16　配列の使用状況

Sieve ほどではないにせよ，BooleanStrategy の恩恵を受けられている．

Int8Strategy が多く用いられているのは Havlak, Json, Permute, Queens, dht11 の 5 つだが，中でも dht11 は全体に占める割合が大きく，生存している割合も高いことから，Int8Strategy の実装により大きくメモリ使用量が減少していることが分かる．

Float64Strategy が用いられていたのは主に List, Mandelbrot, NBody の 3 つだが，いずれもプログラム中で `Array` オブジェクトはほとんど確保されておらず，メモリ使用量への影響は限定的だった．

5.4　考察

5.2 節と 5.3 節の結果から，提案手法により，確かにメモリ使用量が減少することが明らかになった．一方で，実行時間の計測結果は，実装するストラテジーの数を増やせば増やすほど，ストラテジーによる処理の分岐にかかるコストが増加し，パフォーマンスに悪影響を与えることを示唆している．このため，導入するストラテジーの選択は慎重に行わねばならない．

Float64Strategy は，JavaScript の仕様で定められた数値をすべて表現できるストラテジーでありながらも，Int8Strategy と Int16Strategy が存在している環境下では，ほとんど用いられることがないことが分かった．eJSVM のようなタグ付きポインターを利用している VM では，4.2 節で示したように，Float64Strategy の実装により，配列からの値の取り出しがメモリアロケーションが発生する操作になってしまう．メモリ使用量の減少という結果が得られないのであれば，実装しないほうが良いと考えられる．

BooleanStrategy の実装については注意が必要である．真理値の配列を扱うプログラムでは，そのメモリ削減効果は大きいが，その代わりに 3% 程度のパフォーマンスオーバーヘッドが発生するためである．これは，バイト境界を無視した値の取り出しと書き込みのために必要なビット演算のコストが大きいためだと考えられる．

EmptyStrategy や Int8Strategy，Int16Strategy は軽量ながらも，メモリ使用量の削減には大きく貢献しており，バランスの取れたストラテジーである．

以上のことから，ポインタータギングを行う JavaScript VM では，Int8Strategy と Int16Strategy，必要に応じて BooleanStrategy を導入することが適切であるといえる．

6　関連研究・技術

ECMAScript 2015 以降の JavaScript には，`TypedArray` と呼ばれるバイナリデータを扱うためのオブジェクトが定義されている [3]．`TypedArray` は，`Uint8Array` や `Float64Array` などの要素のデータ型を明示したオブジェクトの総称で，通常の `Array` オブジェクトに似たインターフェースが提供されている．データ型が決まっているためメモリ効率は高いが，C などの配列と同じく要素数の変更は許されておらず，通常の `Array` オブジェクトに利便性の点で劣る．なお，eJSVM では `TypedArray` は利用できない．

高速な JavaScript VM として知られる V8 では，数値のみを対象としたストレージストラテジーとして，`ElementsKind` を配列に対して設定している．`ElementsKind` は，空要素が含まれる場合と含まれない場合で別の `ElementsKind` を定義しているという点で，全てのストラテジーが空要素に対応可能な本研究の実装と異なっている [5].

7 おわりに

本研究では，ストレージストラテジーを応用し，JavaScript VM のメモリ使用量を削減するための手法を提案した．この手法は，値の大きさに応じたストラテジーを自動的に適用し，配列の要素を必要に応じて 1 ワードより小さなメモリ領域で表現することで，積極的なメモリ最適化を実現している．また，提案手法の設計では，JavaScript に特有である空要素を含む配列が，各ストラテジーによって対応されるべきであることを指摘した．

提案手法を組み込み向け JavaScript VM である eJSVM に実装し，既にメモリ最適化が進んでいる処理系においても，手法が有効であることを示した．さらに，各ストラテジーの効果を定量的に測定し，ストラテジーの設計者が実装すべきストラテジーについての指針を提示した．

提案手法は，用意した 15 個のベンチマークプログラムのうち，数値や真理値といったプリミティブからなる配列を扱っている，4 個のメモリ使用量を大幅に削減することに成功した．残り 11 個のプログラムの多くで利用されているオブジェクトの配列に対して有効な手法を考案することは今後の課題である．

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] ECMA International: *Standard ECMA-262 - ECMAScript Language Specification 5.1 Edition*, 2011.

[3] ECMA International: *Standard ECMA-262 - ECMAScript 2015 Language Specification*, 2015.

[4] Gudeman, D.: Representing Type Information in Dynamically Typed Languages, Technical Report TR93-27, The University of Arizona, 1993.

[5] 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.

[6] 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.

[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.