JavaScript における静的最適化と動的性質の統合と評価

笠 将稀　齋藤 匡　岩崎 英哉

本研究は，クラス構文およびクラスフィールド宣言を用いる JavaScript プログラムに対し，静的情報を活用する最適化と，JavaScript の動的性質との両立について検討する．先行研究では，隠れクラスに基づく属性への固定オフセットによる直接アクセスと，隠れクラスを事前に構築する最適化を示した上で，メソッド持ち出しや属性削除などの動的挙動により，固定オフセットによる直接アクセスが不正な参照を起こさないようにする対策が提案されている．本研究では，この先行研究で採用されたメソッド持ち出しを検出した時点で即座に脱特殊化する eager 方式に対し，持ち出されたメソッドが実際に呼ばれた時点で脱特殊化する lazy 方式により，不要な脱特殊化を削減する設計を検討した．また，属性削除に対しては削除された属性に印をつけるマーカ方式に対し，削除された属性への固定オフセットによる直接アクセスを禁止する脱特殊化方式を検討した．これらの方式を QuickJS に実装し，実行時間，メモリ使用量，およびラッパ生成や脱特殊化などの動的事象の発生状況に関する定量的評価を行った．その結果，lazy 方式は不要な脱特殊化済み実体の生成を抑制できたが，eager 方式に対する実行時間上の優位性は限定的であった．また，脱特殊化方式では，属性削除が頻発する場合に，実行時コストが直接アクセスの効果を上回ったと推測される．

1　はじめに

JavaScript の標準的な仕様である ECMAScript には，ES2015 のクラス構文 [5] や ES2022 のクラスフィールド宣言 [6] など，クラスベースのオブジェクト指向言語に類似した構文が導入されている．これらの構文はクラスベースの言語から借用したものの，JavaScript 自体は依然としてプロトタイプベース言語である．したがって，クラスベースの静的言語に対する最適化技術を JavaScript に単純に適用する試みは，複数の問題を引き起こす．

Saito らの先行研究 [11] は，クラスフィールドを用いるプログラムを対象に，属性を固定オフセットで参照するメソッド特殊化と，属性追加に伴う隠れクラス (Hidden Class，以下 HC) [1] [4] の遷移を削減する事前構築隠れクラス (Prior-built HC，以下 PBHC) を提案した．また，最適化を安全に適用するうえで問題となるメソッド持ち出し，属性削除，動的継承に対し，複数の対処法を示した．これらの対処法の一部は JavaScript エンジン QuickJS [3] へ実装されたが，まだ実装されていない対処法もあり，実装に基づく定量的な比較評価はまだなされていない．

本研究では，メソッド持ち出しと属性削除に対する未実装方式を QuickJS へ実装し，既実装の方式と定量的な比較を行う．メソッド持ち出しについては，それを検出した時点で脱特殊化する eager 方式 (既実装) に対し，持ち出されたメソッドが実際に呼ばれた時点で脱特殊化する lazy 方式を実装する．属性削除については，削除属性に印をつけるマーカ方式 (既実装) に対し，削除された属性を固定オフセットで参照するメソッドを，通常の属性アクセスへ戻す脱特殊化方式を実装する．既存方式と実行時間，メモリ使用

Integration and Evaluation of Static Optimization and Dynamic Features in JavaScript
Masaki Ryu, 明治大学大学院理工学研究科情報科学専攻, Computer Science Course, Graduate School of Science and Technology, Meiji University.
Tadashi Saito, 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, Dept. of Computer and Network Engineering, Graduate School of Informatics and Engineering, The University of Electro-Communications.
Hideya Iwasaki, 明治大学理工学部情報科学科, Dept. of Computer Science, School of Science and Technology, Meiji University.

```
1 function Point(x, y) {
2   this.x = x;
3   this.y = y;
4 }
5 Point.prototype.manhattan = function() {
6   return this.x + this.y;
7 };
```

図 1　ES2015 以前のオブジェクト定義

量，動的処理の発生回数を比較し，各方式の特性を評価する．

以下，第 2 節で JavaScript のクラス構文，第 3 節で先行研究を説明する．第 4 節で提案方式の実装，第 5 節で評価を述べ，第 6 節で本研究をまとめる．

2　JavaScript におけるクラス構文

本節では，本研究の前提となる JavaScript のクラス構文とクラスフィールド (以下 CF) 宣言について説明する．

2.1　ES2015 クラス

JavaScript のオブジェクトは，別のオブジェクトを参照する内部的なプロトタイプを持つ．属性を参照すると，まず対象オブジェクト自身の中の属性を探索し，属性が存在しなければプロトタイプを順にたどる．この探索経路をプロトタイプチェーンと呼び，JavaScript の継承はこの仕組みにより実現される．

ES2015 以前は，図 1 のようにコンストラクタ関数で属性を初期化し，そのプロトタイプへメソッドを登録していた．これにより，各インスタンスは属性値を個別に保持しながら，メソッドを共有できる．

ES2015 [5] では，図 2 のように従来のコンストラクタ関数とプロトタイプによる定義を簡潔に表すクラス構文が導入された．図 2 の `constructor` メソッドはインスタンス生成時の初期化を行い，クラス本体に記述したメソッドはインスタンス間で共有される．ただし，クラス構文が導入されても JavaScript はプロトタイプベースの言語であり，実行時のメソッド探索は引き続きプロトタイプチェーンに基づく．

```
1 class Point {
2   constructor(x, y) {
3     this.x = x;
4     this.y = y;
5   }
6   manhattan() {
7     return this.x + this.y;
8   }
9 }
```

図 2　ES2015 のクラス構文の例

```
1 class Point {
2   x;
3   y;
4   constructor(x, y) {
5     this.x = x;
6     this.y = y;
7   }
8   manhattan() {
9     return this.x + this.y;
10  }
11 }
```

図 3　ES2022 クラスフィールド宣言の例

2.2　ES2022 クラスフィールド構文

ES2022 [6] では，インスタンスが持つ属性をクラス本体に記述する CF 宣言が標準化された．図 3 では，`x` と `y` をパブリックフィールドとして宣言している．各フィールドは宣言順にインスタンスへ追加され，初期化子がある場合はインスタンス生成時に評価される．

CF には，パブリックフィールドのほか，クラス外部から直接参照できないプライベートフィールドと，クラス自体が保持する静的フィールドがある．本研究では，パブリックフィールドを最適化の対象とする．

3　JavaScript エンジン最適化の先行研究

本節では，CF 宣言を利用した先行研究 [11] について述べる．先行研究は，メソッド特殊化と事前構築隠れクラスを QuickJS へ実装し，JavaScript の動的性質によって生じる問題とその解決策を示している．

3.1 最適化手法

3.1.1 背景

図 3 に示したクラス Point から生成されるオブジェクトは，生成時に属性 x と y をこの順序で持つ．また，JavaScript のオブジェクトは，生成後も属性を追加・削除できる．

多くの JavaScript エンジンは，オブジェクトの属性配置を HC で表現する．HC は，属性名とオブジェクト内部における属性値の配置 (オフセット) との対応を保持し，同じ属性配置を持つオブジェクト間で共有される．属性が追加・削除されると，オブジェクトは新しい HC を参照する．これを HC 遷移と呼ぶ．HC は属性の追加順序に依存するため，同じ属性集合を持つオブジェクトでも追加順序が異なれば別の HC を参照する．

CF 宣言を用いると，同じクラスのインスタンスは生成時に同じ属性集合と追加順序を持つ．先行研究はこの性質を利用し，属性アクセスとオブジェクト生成を静的に最適化する．

3.1.2 メソッド特殊化

通常の属性アクセス (以下，通常アクセス) では，オブジェクトの HC を取得し，属性名からオフセットを求めた後，その位置の値を読み書きする．メソッド特殊化は，クラス内で定義されたメソッドにおいて，CF へのアクセスを固定オフセットにより直接行う (以下，直接アクセス) ．例えば図 3 では，x と y のオフセットをあらかじめ決定できるため，manhattan 内の 2 回の属性アクセスを直接行い，HC を利用した探索を省略できる．

特殊化の対象は，クラス内で定義されたメソッドとコンストラクタから，静的に特定できる CF へアクセスする場合である．CF にない属性や属性名が実行時まで決まらないアクセスには，HC を用いた通常アクセスを用いる．以下，直接アクセスを含む関数オブジェクトを特殊化メソッドと呼ぶ．

3.1.3 事前構築隠れクラス (PBHC)

通常は，インスタンスへ CF を追加するたびに HC 遷移が発生する．これに対し，PBHC は，CF 宣言から最終的な属性配置情報を持つ HC を先に構築し，生成直後のインスタンスへ一度に関連付ける．図 3 の Point では，x だけを持つ一時的な HC を経由せず，x と y を持つ PBHC を最初から利用する．

```
1 let p = new Point(12, 34);
2 let q = {};
3 q.f = p.manhattan;
4 q.f();
```

図 4 単一メソッド持ち出しの例

PBHC により，中間 HC の生成と HC 遷移の処理を削減できる．また，すべての CF を格納する領域を先に確保できるため，コンストラクタによる初期化にも固定オフセットの直接アクセスを適用できる．

3.2 動的性質に起因する課題

メソッド特殊化は，実行時にレシーバの属性配置が想定どおりであることを前提とする．しかし，JavaScript ではメソッド，属性，継承関係を動的に変更できるため，先行研究はこれらに起因する 3 つの問題を扱っている．そのうち，ここでは本研究に関係する 2 つの問題に絞って説明する．

3.2.1 メソッド持ち出し

JavaScript のメソッドは第一級オブジェクトであり，変数や別のオブジェクトへ持ち出して代入できる．図 4 では，Point の特殊化メソッドを想定とは別のオブジェクト q の属性 f へ代入し (3 行目) ，q をレシーバとして呼び出している (4 行目) ．f の実体は manhattan であり，このまま直接アクセスを行うと，q には存在しない属性配置へアクセスしてしまう．

図 4 のように特殊化メソッドを個別に持ち出す場合を単一メソッド持ち出し，特殊化メソッドを含むプロトタイプオブジェクト全体を持ち出す場合を複数メソッド持ち出しと呼ぶ．後者では，プロトタイプに登録された複数の特殊化メソッドが想定外のオブジェクトをレシーバとして呼び出される可能性がある．

3.2.2 属性削除

JavaScript では，CF として宣言した属性も削除できる．図 5 のように y を削除すると，manhattan で y に直接アクセスする命令は誤りとなる．さらに，削除済み領域を詰めるコンパクションが行われると，残っ

```
1  delete p.y;
2  p.manhattan();
```

図 5　属性削除の例

た属性のオフセットも変化し得る．

3.3　先行研究における解決手法

3.3.1　メソッド持ち出しへの対処

持ち出された特殊化メソッドを，直接アクセスを通常アクセスへ置き換えた別の関数オブジェクトへ変換する．先行研究はこの変換を脱特殊化と呼ぶ．元の特殊化メソッドは想定通りのインスタンスから呼び出すために残し，持ち出された場合のみ脱特殊化済みメソッドを生成し，それを返す．複数メソッド持ち出しでは，プロトタイプオブジェクトを複製し，内部の特殊化メソッドをすべて脱特殊化済みメソッドにする．

脱特殊化は，持ち出しの検出時に変換する eager 方式と，持ち出されたメソッドが実際に呼び出された時に変換する lazy 方式があるが，先行研究において実際に実装・評価が行われたのは eager 方式のみである．また，内部的には特殊化されたオブジェクトと脱特殊化されたオブジェクトが同時に存在し得るため，JavaScript から見た同一性を保ち，一方への属性変更を他方へ反映する仕組みが必要である．先行研究は，標準機能のひとつである Proxy を用いてこの対応を管理する方法を示している．

3.3.2　属性削除への対処

先行研究は属性削除に対して 2 つの方法を示している．1 つ目のマーカ方式は，削除済みであることを示す印を削除属性のオフセットへ付加し，直接アクセス時にその情報を検査する．特殊化を維持できるという長所がある一方，直接アクセスごとの検査が必要であり，コンパクションで属性位置が変わる場合には直接アクセスとの整合性を保たなければならないという短所もある．

2 つ目の脱特殊化方式は，属性削除によって直接アクセスの前提が崩れた特殊化メソッドを通常アクセスへ戻す脱特殊化を行う．これによりコンパクション後も安全に実行できるが，脱特殊化の処理と，その後の通常アクセスによるコストが生じる．

先行研究はマーカ方式を実装しているが，脱特殊化方式については実装・評価を行っていない．

3.4　QuickJS への実装

先行研究は，メソッド特殊化，PBHC，および動的性質への対処 (eager 方式とマーカ方式) を QuickJS (2024-01-13) へ実装した．QuickJS は構文解析と同時にバイトコードを生成する軽量なインタプリタであり，HC に相当する既存の属性配置情報を持つ．

メソッド特殊化のため，固定オフセットの直接アクセスを行う命令が複数追加された．これらはメソッドから自らの属性へアクセスする場合に限って生成され，そのオフセットは PBHC の配置情報から得られる．メソッド持ち出しを検出した時は，eager 方式で脱特殊化する．また，重複した脱特殊化を防ぐため，元のオブジェクトと脱特殊化済み関数オブジェクトの対応も保持する．

属性削除には，QuickJS の HC が既に持つ削除マーカを利用する方式が採用された．これにより新たな削除用データ構造は不要であったが，整合性維持のためにコンパクションを一部無効化している．

4　本研究の位置付けと実装

本研究は，先行研究で実装・評価が行われていない，メソッドが実際に呼び出されるまで脱特殊化を遅延する lazy 方式と，属性削除時に対象メソッドを脱特殊化する脱特殊化方式を実装する．さらに，これらを先行研究の eager 方式，マーカ方式と比較し，不要な脱特殊化の削減効果と，属性削除時の実行時コストを評価する．

4.1　メソッド持ち出しに対する対処: lazy 方式

eager 方式は，メソッド持ち出しを検出した時点で即座に脱特殊化を行う．しかし，そのメソッドが呼び出されなかった場合，脱特殊化は無駄なコストになる．これに対し lazy 方式は，メソッドが持ち出された時には脱特殊化を行わず，それが呼ばれた時に初めて脱特殊化を行うため，不要な脱特殊化を削減できる．

3.2.1 節で述べたように，メソッド持ち出しには単一メソッド持ち出しと複数メソッド持ち出しがある．以下では，これらのそれぞれに対し，lazy 方式の実装方針を述べる．

4.1.1 単一メソッド持ち出し

単一メソッド持ち出しとは，特殊化メソッド単体を別のオブジェクトに保存する操作である．例えば，`q.f = p.manhattan` によって `p.manhattan` を持ち出す場合が該当する．

lazy 方式では，メソッドを持ち出した時点では脱特殊化を行わず，元の特殊化メソッドと特殊化時に参照した HC の組を保持するような，呼び出し可能な Proxy を返す．以下，この Proxy を関数用ラッパと呼ぶ．同じ特殊化メソッドと HC の組に対する関数用ラッパはキャッシュし，繰り返し持ち出した場合にも同一の関数用ラッパを返す．これにより，同じ特殊化メソッドと HC の組に対する関数用ラッパの重複生成を避ける．同じメソッドであっても HC が異なれば，固定オフセットに対応する属性が異なり得るため，別の関数用ラッパとする．

関数用ラッパが呼び出されると，特殊化メソッドと HC の組に対応する脱特殊化済み実体を検索する．初回呼び出し時には実体が存在しないため，元のバイトコードを複製し，固定オフセットを用いる直接アクセス命令を属性名による通常アクセス命令へ変換する．その後，変換後のバイトコードを保持する脱特殊化済み実体を生成してキャッシュし，持ち出し後のレシーバと引数を渡して呼び出す．2 回目以降はキャッシュ済みの実体を再利用する．元の特殊化メソッドは変更しないため，想定通りのレシーバが指定されたメソッド呼び出しでは特殊化されたバイトコードを引き続き利用できる．

この方式では，持ち出されたメソッドが呼び出されなければ，関数用ラッパの生成だけが行われ，脱特殊化のコストは発生しない．また，脱特殊化は特殊化メソッドと HC の組ごとに初回のみ行われる．

4.1.2 複数メソッド持ち出し

複数メソッド持ち出しは，プロトタイプオブジェクト (以下，単にプロトタイプと呼ぶ) を持ち出す操作である．例えば，`po = Point.prototype` によってプロトタイプを変数へ保存する場合が該当する．

eager 方式では，プロトタイプが持ち出された時点で，そこに含まれる特殊化メソッドをすべて脱特殊化する．これに対して lazy 方式では，元のプロトタイプを対象とする Proxy を返し，個々のメソッドが必要になった時点まで脱特殊化を遅延する．以下，これをプロトタイプ用ラッパと呼ぶ．同じプロトタイプに対するプロトタイプ用ラッパはキャッシュし，繰り返し持ち出した場合にも同一のプロトタイプ用ラッパを利用する．

プロトタイプ用ラッパを介して属性が読み出されると，元のプロトタイプから値を取得する．取得した値が特殊化メソッドである場合は，取得した値の代わりに 4.1.1 節で述べた関数用ラッパを返す．通常の値であればそのまま返し，getter には実際の属性アクセスで指定されたレシーバを引き継ぐ．したがって，メソッドを参照しただけでは関数用ラッパの生成にとどまり，脱特殊化済み実体は生成されない．関数用ラッパが実際に呼び出された時点で，そのメソッドだけを脱特殊化し，2 回目以降はキャッシュ済みの脱特殊化済み実体を再利用する．

4.1.3 ラッパを介した操作

ラッパを介した属性の書き込み，削除，属性記述子の取得と変更，および拡張禁止の操作は，元のプロトタイプへ反映する．属性記述子の値，getter，または setter が特殊化メソッドである場合は，それらをラッパへ置き換えて公開する．また，属性の変更時には，ラッパが公開する属性情報と元のプロトタイプの状態が矛盾しないように同期する．これにより，通常の属性操作との整合性を保ちながら，脱特殊化のコストは実際に参照され，かつ呼び出されたメソッドに限定できる．

4.2 属性削除に対する対処

属性が削除されると，特殊化メソッド内の直接アクセス命令が削除済み属性が配置されていた場所 (空きスロット) を参照してしまう．さらに，空きスロットを詰めて，他の属性のオフセットも変化した場合，削除された属性を参照しない特殊化メソッドでも誤った位置へのアクセスが起こる．

本研究の脱特殊化方式では，削除属性を参照するメソッドのみを脱特殊化し，それ以外の特殊化メソッドはそのまま維持する．このため，属性削除時に対象オブジェクトと同じ属性配置を前提とする特殊化メソッドが検出された場合は，残存属性のオフセットを変更するコンパクションは行わず，削除済みスロットを空きとして残す．また，属性の削除時に対象オブジェクトのプロトタイプチェーン上のメソッドを探索する．各メソッドについて，特殊化時に用いた HC から削除属性のオフセットを求め，そのオフセットを直接参照する命令が含まれるかを検査し，あれば，そのメソッドに含まれるすべての直接アクセス命令を属性名による通常アクセス命令へ変換する．変換対象外のメソッドは特殊化されたまま残るため，削除属性と無関係なメソッドの特殊化を維持できる．

属性削除前に持ち出されたメソッドへの参照をすべて追跡し，脱特殊化済みの別実体へ差し替えることには大きなコストがかかる．そこで，本実装では既存メソッドのバイトコードを直接変換する．これにより，プロトタイプ，変数，関数用ラッパなどから同じメソッドが参照されている場合も，削除後は通常の属性アクセスを利用できる．一方で，同じメソッドを利用する他のインスタンスでも特殊化が失われるという欠点がある．

5 評価

QuickJS に対する本実装の評価には，Apple M4 を搭載した MacBook Air を用いた．論理コア数は 10，メモリは 24 GB である．OS は macOS 26.6 (Build 25G72) ，C コンパイラは Apple Clang 21.0.0 を使用した．QuickJS の版は 2026-06-04 である．

評価対象は次の 5 実装である．

- $\mathsf{QJS_o}$: 元々の実装
- $\mathsf{QJS_{em}}$: eager 方式 + マーカ方式
- $\mathsf{QJS_{lm}}$: lazy 方式 + マーカ方式
- $\mathsf{QJS_{ed}}$: eager 方式 + 脱特殊化方式
- $\mathsf{QJS_{ld}}$: lazy 方式 + 脱特殊化方式

$\mathsf{QJS_o}$ に対する追加行数は，$\mathsf{QJS_{em}}$ が 2326 行，$\mathsf{QJS_{lm}}$ が 2419 行，$\mathsf{QJS_{ed}}$ が 2523 行，$\mathsf{QJS_{ld}}$ が 2616 行である．追加コードの内訳は，全方式共通が 2306 行，eager 方式固有が 20 行，lazy 方式固有が 113 行，脱特殊化方式固有が 197 行である．

本評価では，ARES-6 [9]，JetStream 2 [2]，Octane 2 [7]，SunSpider 1.0.2 [8] から選んだ既存 13 プログラムと，自作した deletion を使用した．既存プログラムには，Saito ら [11] が Lebab [10] と手作業によりクラス構文とクラスフィールド宣言を導入した版を用いた．

表 1 に各プログラムの概要を示す．既存の 13 プログラムでは属性削除が行われなかった．そこで，属性削除への対処を評価するため，deletion を追加した．deletion は 12 個の CF を持つオブジェクトを 100 万個生成し，各オブジェクトに対して以下を行う．8 個のフィールドを参照するメソッドと残り 4 個を参照するメソッドを実行し，前者が参照する 8 個のフィールドを削除した後，同じ 2 メソッドを再実行する．したがって，1 実行当たり 800 万回の属性削除と 400 万回のメソッド呼び出しが行われる．deletion 以外のプログラムについては，属性削除は起こらなかったため，実行時間とメモリ使用量は，$\mathsf{QJS_o}$，$\mathsf{QJS_{ed}}$，$\mathsf{QJS_{ld}}$ の結果だけを示している．また，deletion については，メソッド持ち出しは起こさないため，$\mathsf{QJS_o}$，$\mathsf{QJS_{lm}}$，$\mathsf{QJS_{ld}}$ の結果を示した．

5.1 メソッド持ち出し

表 2 に，$\mathsf{QJS_{lm}}$ で観測したメソッド持ち出しに関する結果を示す．W，C，D，P はそれぞれ，関数用ラッパ生成数，同ラッパを経由した呼び出し数，脱特殊化済み実体生成数，プロトタイプ用ラッパ生成数を表し，D/W は実体生成に至った割合を表す．プロトタイプ用ラッパから個別のメソッドを取得した場合にも関数用ラッパが生成されるが，その回数も W に含まれる．

全体では関数用ラッパが 20 個生成され，そのうち 8 個が脱特殊化済み実体の生成に至った．したがって，lazy 方式は 12 個，すなわち 60.0%の実体生成を回避した．特に ML では 8 個中 7 個，GB Emulator では 7 個中 4 個の実体生成を回避した．Babylon では関数用ラッパが 1 個生成されたが呼び出されなかった．一方，Air，DeltaBlue，3d-raytrace では生成された

表 1　ベンチマークのプログラム (*は Lebab 利用を示す)

プログラム名	概要	ソース
Air	WebKit エンジンの JIT の一部	ARES-6
Babylon	JavaScript パーサ	ARES-6
ML	フィードフォワードニューラルネットワーク	ARES-6
async-fs	ファイルシステムのエミュレータ	JetStream 2
DeltaBlue*	制約解決アルゴリズム	Octane 2
EarleyBoyer*	Scheme 由来の 2 つのベンチマーク	Octane 2
GB Emulator*	携帯型ゲーム機のエミュレータ	Octane 2
Raytrace*	レイトレーサ	Octane 2
Richards*	タスクディスパッチャのシミュレーション	Octane 2
Splay*	スプレー木の操作	Octane 2
3d-raytrace*	単純なレイトレーサ	SunSpider 1.0.2
binary-trees*	二分木の生成	SunSpider 1.0.2
nbody*	N 体問題のシミュレーション	SunSpider 1.0.2
deletion	クラスフィールドの削除	自作

表 2　lazy 方式におけるラッパ生成，呼び出し，および脱特殊化の発生回数
(W または P が 0 より大きいプログラムのみ掲載)

プログラム名	関数用ラッパ 生成数 W	呼び出し数 C	実体生成数 D	D/W (%)	プロトタイプ用ラッパ 生成数 P
Air	2	84200	2	100.0	0
Babylon	1	0	0	0.0	2
ML	8	840	1	12.5	3
DeltaBlue	1	268100	1	100.0	1
GB Emulator	7	297900	3	42.9	1
Splay	0	0	0	–	2
3d-raytrace	1	90000	1	100.0	0

関数用ラッパがすべて実体生成に至った．これらのプログラムでは関数用ラッパが多数呼び出されるが，脱特殊化済み実体は再利用されるため，実体生成数は増加しない．

5.2　実行時間

図 6 に実行時間を示す．実行時間は，ベンチマークが内部で測定した経過時間を用いた．各実装について 1 回の予備実行を行った後，5 回測定した．実行順序の偏りを抑えるため，プログラムと測定回ごとに 5 実装の実行順序を巡回させた．既存ベンチマークでは，各回の反復結果を平均した値を使用した．

図 6 の各値は，5 回の平均を $\mathsf{QJS_o}$ の平均で除したものである．Average は deletion を除く 13 プログラムの比の平均である．個別プログラムの誤差線は，各実装の 5 測定値の標本標準偏差を，同じプログラムの $\mathsf{QJS_o}$ の平均で正規化した値を示す．Average は $\mathsf{QJS_{ed}}$ が 97.8%，$\mathsf{QJS_{ld}}$ が 97.4%であった．最適化を施した 2 実装の差は小さい．

個別の結果では，Raytrace が約 16%，binary-trees が約 14%，ML が約 5%，DeltaBlue が約 6%短縮された．一方，Splay は約 8%から 10%，GB Emulator は約 6%増加した．これらの傾向は 2 方式でおおむね共通している．

deletion では，$\mathsf{QJS_{lm}}$ が 77.9%で $\mathsf{QJS_o}$ より高速であったのに対し，$\mathsf{QJS_{ld}}$ は 167.2%であった．両方式ともコンパクションを抑止するため，この差は，脱特殊化方式で属性削除のたびに発生する実行時コストによるものと考えられる．

5.3　メモリ使用量

図 7 にメモリ使用量を示す．メモリ使用量の測定では，実行時間の測定と同一のソースから 5 実装を個別に構築した．各プログラムを実装ごとに 5 回実行

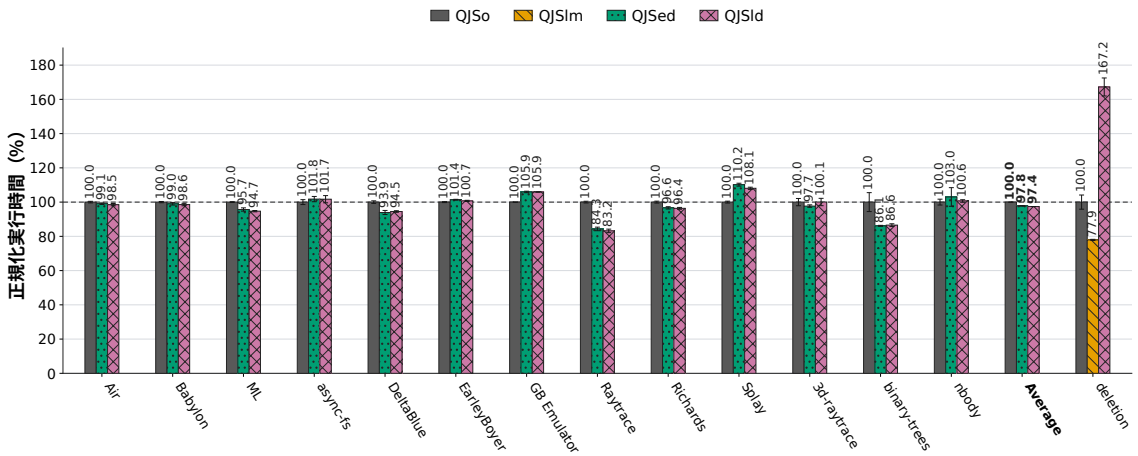

図 6　実行時間の比較

し，実行終了時点における JavaScript エンジン内部のメモリ使用量を測定した．この値は，実行ファイルや共有ライブラリを含む OS 上のプロセス全体の使用量とは異なる．図 7 の値は，各実装の 5 回の平均値を，同じプログラムにおける $\mathsf{QJS_o}$ の 5 回の平均値で除した結果である．Average は deletion を除く 13 プログラムの比の平均である．

最も大きい増加は EarleyBoyer の 35.9%であった．Air，ML，async-fs，3d-raytrace，binary-trees，nbody でも 20%を超えた．一方，Babylon，DeltaBlue，GB Emulator の増加は約 1%から 5%であった．Raytrace，Richards，Splay，deletion も同じ範囲に収まった．

eager 方式と lazy 方式の差はプログラムによって異なる．ML では lazy 方式が eager 方式より約 4.6 ポイント少なく，GB Emulator と Splay でも lazy 方式の方が少なかった．これは，lazy 方式が持ち出されたメソッドを実際に呼び出すまで脱特殊化済み実体を生成しないためであると言える．一方，3d-raytrace では lazy 方式が約 0.3 ポイント多く，ラッパの管理によるコストが現れている．

マーカ方式と脱特殊化方式の値は全プログラムで同一であり，今回のメモリ使用量では属性削除方式による差は確認できなかった．

6　おわりに

本研究では，先行研究 [11] によるメソッド特殊化と PBHC を基盤として，JavaScript の動的性質に対する方式を QuickJS へ実装した．メソッド持ち出しでは eager 方式と lazy 方式，属性削除ではマーカ方式と脱特殊化方式を比較した．

deletion を除く 13 プログラムでは，2 方式の実行時間の平均は元実装の 97.4%から 97.8%であり，方式間の差は小さかった．lazy 方式は 20 個の関数用ラッパのうち 12 個で脱特殊化済み実体の生成を回避したが，既存プログラムではメソッド持ち出しが少なく，また属性削除が行われないため，実行時間への効果は限定的であった．一方，deletion ではマーカ方式が元実装の 77.9%であったのに対し，脱特殊化方式は 167.2%となった．属性削除時の実行時コストが，直接アクセスの効果を上回ったと推測される．

QuickJS が管理するメモリ使用量の 13 プログラムにおける平均は，元実装に対して 114.5%から 115.3%であった．マーカ方式と脱特殊化方式の値は一致し，属性削除方式による差は確認されなかった．また，lazy 方式は ML，GB Emulator，Splay では eager 方式よりメモリ使用量を抑えたが，方式間の差が小さいプログラムも多かった．

以上から，lazy 方式は不要な実体生成を抑制できるが，eager 方式に対する実行時間上の優位性は限定的であり，現在の脱特殊化方式は属性削除が頻発する

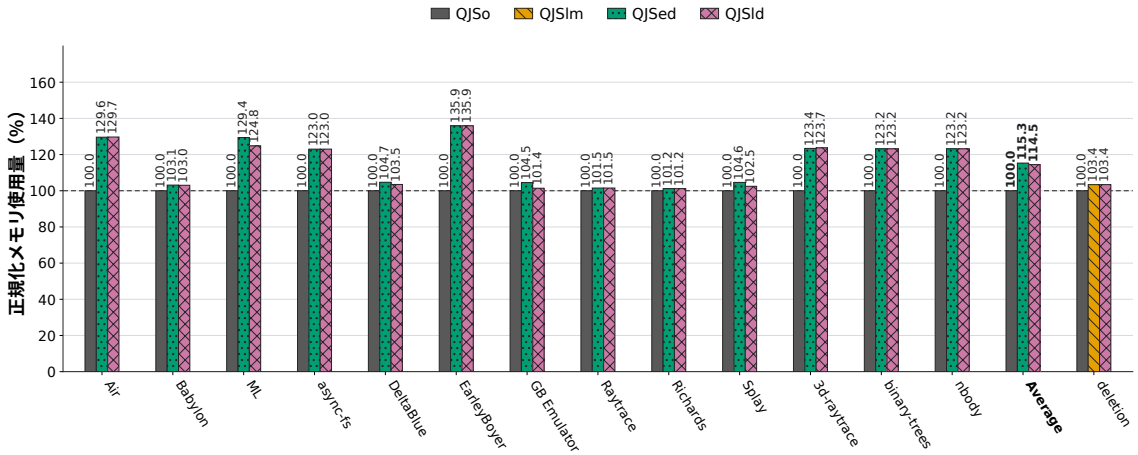

図 7　メモリ使用量の比較

場合のコストが大きいことが分かった．今後は，削除時の実行時コストと属性配置維持のコストを削減し，動的事象の発生条件を制御した評価を行う．また，実行時間やメモリ使用量が増加した原因を分析し，動的継承を含む他の動的性質についても統合的に評価する必要がある．

参 考 文 献

[1] Ahn, W., Choi, J., Shull, T., Garzarán, M. J., and Torrellas, J.: Improving JavaScript performance by deconstructing the type system, *ACM SIGPLAN Notices*, Vol. 49, No. 6(2014), pp. 496–507.

[2] Barati, S. and other contributors: JetStream2. Available at `https://github.com/WebKit/WebKit/tree/ea2c2766/PerformanceTests/JetStream2` [Online; accessed 2026-01-05].

[3] Bellard, F. and Gordon, C.: QuickJS Javascript Engine. Available at `https://bellard.org/quickjs/` [Online; accessed 2025-11-25].

[4] Chambers, C., Ungar, D., and Lee, E.: An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes, *ACM Sigplan Notices*, Vol. 24, No. 10(1989), pp. 49–70.

[5] ECMA International: ECMAScript 2015 Language Specification – ECMA-262 6th Edition, 2015. Available at `https://262.ecma-international.org/6.0/` [Online; accessed 2025-11-13].

[6] ECMA International: ECMAScript 2022 Language Specification – ECMA-262 13th Edition, 2022. Available at `https://262.ecma-international.org/13.0/` [Online; accessed 2025-11-13].

[7] Hablich, M. and other contributors: Octane. Available at `https://github.com/chromium/octane` [Online; accessed 2026-01-05].

[8] Lima, C. and other contributors: SunSpider. Available at `https://github.com/WebKit/WebKit/tree/cb2f5004/PerformanceTests/SunSpider` [Online; accessed 2026-01-05].

[9] Meurer, B. and other contributors: ARES-6. Available at `https://github.com/bmeurer/ARES-6` [Online; accessed 2026-01-05].

[10] Saarsoo, R. and other contributors: Lebab. Available at `https://github.com/lebab/lebab` [Online; accessed 2026-01-12].

[11] Saito, T. and Iwasaki, H.: Integrating Static Optimization and Dynamic Nature in JavaScript, *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2025, pp. 41–53.