小型デバイス用 JavaScript エンジンの効率化

坂本拓也[†1]　　伊藤栄信[†1]　　二村和明[†1]

さまざまなモノがセンサーを備えるとともにインターネットに接続して情報交換する Internet of Things，特にそのサービスの実現に Web 技術を使う Web of Things が注目されている．モノに搭載するデバイスは，これまで Web 技術の中心だった PC やスマートフォンなどの高スペックデバイスに比べて，コストや消費電力の観点から低スペックでの動作が要求される．そこで，Web 技術である JavaScript を動作させるマイコンの開発などが進められている．
本稿では，低スペックである小型デバイス上でセンサーや通信などのイベントに対応してサービスを提供するイベントドリブンモデルを想定した上で，そのモデルに適した高速化手法を提案する．本提案手法は，イベントに応じて繰り返して実行される JavaScript で記述された関数の呼び出しを効率化するために，その関数の 1 度目のパーズ・実行時にパーズ結果から得られる中間コードを記憶しておき，2 度目以降の実行時に，それを利用してパーズ処理を省略することを可能とする．また，本提案手法を実装して，関数の実行時間を測定・評価し，最大 3.6 倍に高速化されることを確認した．

JavaScript Engine Optimization for Small Devices

TAKUYA SAKAMOTO[†1]　HIDENOBU ITO[†1]　KAZUAKI NIMURA[†1]

1. はじめに

さまざまなモノがセンサーを備えるとともにインターネットに接続して情報交換する Internet of Things により，モノを識別したり，モノの状況を把握したり，モノを制御することが可能になってきている．中でもモノと連携するアプリケーションやサービスの開発に Web 技術を使う Web of Things が注目されており，World Wide Web Consortium（W3C）では，ユースケースや要求事項の検討およびその実現方法について議論がなされている[1]．Web 技術を扱える技術者は非常に多いため，アプリケーションやサービスの開発が促進され，その結果，ユーザーはさまざまなメリットを享受できると考えられるためである．

現状，W3C では，モノへのプロキシとして動作する仮想オブジェクトに JavaScript API を使ってアクセスすることを想定しているが，モノが提供するサービス自身の記述手段については対象外である．一方で，たとえば，サーバーが提供するサービスを JavaScript で記述する node.js[2]の普及が進んでいることなどから，Web サービスの記述全般に JavaScript の応用範囲が広がり，モノが提供するサービスについても JavaScript を使用することが予想される．

また，JavaScript はスクリプト言語であり，個々のデバイス向けの実行形式であるバイナリファイルにコンパイルすることなしに，プログラムを実行することが可能である．そのため，さまざまなプラットフォーム上で動作し，プログラムコードの変更が容易であるという柔軟性を備える．この特徴は，プラットフォームに影響されずにアプリケーションを入れ替えることを可能とし，状況にあったサービスの提供の実現につながると考えられる．

しかし，スクリプト言語は一般的にインタープリターを通して実行されるため，事前にコンパイラーにより生成されたバイナリファイルを実行することに比べて動作が遅い．そのため，PC やスマートフォンの Web ブラウザーには，JavaScript を高速で動作させるための技術が搭載されている．たとえば，Google のブラウザーChrome の JavaScript エンジンである Google V8 JavaScript Engine[3]では，full-codegen, crankshaft と呼ばれる Just in Time コンパイラーが搭載されており，単純なインタープリターによる実行に比べてはるかに高速に動作する．しかし，これらの技術は高スペックなデバイスを対象にしており，多くのリソースを要求する．そのため，モノへ搭載するような低スペックの小型デバイス上で，同様の高速化手法を使うことは過剰であり，そのまま適用することは適切ではない．

そこで，我々は低スペックである小型デバイス上でセンサーや通信などのイベントに対応してサービスを提供するモデルを想定した上で，そのモデルに適した高速化手法を提案する．本提案手法では，繰り返して実行される JavaScript コードの呼び出しを効率化するために，その部分の 1 度目のパーズ・実行時にパーズ結果から得られる情報を記憶しておき，2 度目以降の実行時に，その情報を利用してパーズ処理を省略する．また，この手法で効率化されることを実装評価して確認する．

以下，2 章で関連研究，3 章で本システムの提案手法，4 章でその提案手法に沿って行った実装について述べ，5 章

†1（株）富士通研究所
Fujitsu Laboratories Ltd.

でその実装を評価する． 最後に、6 章で結論を述べる。

2. 関連研究

本章では，JavaScript の効率的な実行を実現する既存の JavaScript エンジンの実装および研究について述べる．

JavaScript を動作可能な低スペックのボード Tessel[4]では，JavaScript を Lua に変換し，Lua の Just In Time ランタイム上で実行することで高速化する方法を実装しようとしている[5]．Lua は，JavaScript と同様に変数に型のないスクリプト言語であるが, 実装が比較的容易で高速に動作する．しかし, JavaScript を Lua へ完全に変換する機能の実装は難しいと考えられる．

PC やスマートフォンで使用する Web ブラウザー向けの JavaScript エンジン（たとえば，Chrome ブラウザーの V8，Safari ブラウザーの JavaScriptCore[6], Firefox ブラウザーの SpiderMonkey[7]など）では，Just In Time コンパイラーが実装され，高速化が実現されている．たとえば，V8 では特に繰り返し使われる部分にフォーカスを絞って高度なコンパイルを行う手法や，SpiderMonkey では asm.js と呼ばれる JavaScript に型アノテーションをつけることでコンパイルをしやすくして高速化する手法が実装されている． これらは, 大きなメモリ空間と高速な CPU を活用した方法であり, 我々と対象が異なる．

次のインタープリティション部分を平行に Just In Time コンパイルすることで，コンパイル範囲を限定して高速化する手法[8]がある．また，32 ビットのインストラクションを使う代わりに 16 ビットのインストラクションを使うことで Just In Time コンパイルした結果を小さくする手法[9]がある．さらに，Just In Time コンパイルした結果を再利用する手法[10]がある．他には WebGL/WebCL でアクセラレートする方法[11]や GPGPL によりアクセラレーションする方法[12]がある．しかしながら，これらは我々の対象より高スペックのデバイスを想定している．

3. 提案手法

本章ではモデルと提案手法について説明する．

我々は，Web 技術によって接続されるデバイスとして，センサーなどを備えた小型デバイスを想定している．小型デバイスがセンサーの情報などを消費電力の観点から効率的に処理するためには, それらの情報の生成を受けて, CPU が動作し，処理終了後にはスリープすることが望ましい．したがって，図 1 に示すように，発生したイベントに応じて処理を行うイベントドリブンのモデルを想定した．

まず，小型デバイスは，センサーや通信によって得られた情報をイベントとして受ける．そして，CPU は受け取った情報をもとに計算を行い，その計算結果に応じて，別の機器へ情報通知を行う．たとえば，温度センサーからの入力イベントを受けて，CPU は温度が 30 度を超えているかどうかのしきい値チェックを行い, 超えている場合にのみ, スマートフォンに通知するといったことである．

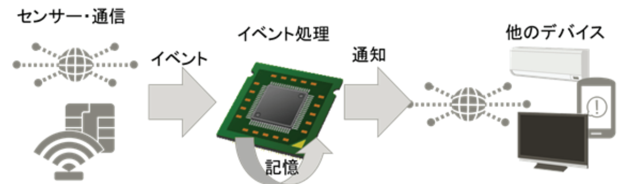

図 1 イベントドリブンモデル
Figure 1 Event driven model.

イベントドリブンモデルを JavaScript で実現するには，イベントに対応づけて関数をコールバックとして渡す方法がある．その結果として，イベントの発生時にその渡した関数が呼び出される．したがって，イベントが起こるごとに関数が呼び出されるため，繰り返して同じ関数が実行されることになる．そこで，我々は，関数単位で実行の効率化を図る手法を提案する．

JavaScript のソースコードを実行する際には，まず，パーズ処理が行われる．パーズは，テキストで記述された JavaScript のソースコードに対して字句解析および構文解析する処理であり時間がかかる．そのため，この時間を短縮することは効果が大きいことが予想される．そこで，関数の実行時にソースコードのパーズ結果から得られる情報を中間コードとして関数ごとに管理し，以降の実行時に，それを利用してパーズ処理を省略する．構成図を図 2 に示す．

本提案手法の構成を以下で説明する．

- イベントループ
  センサーや通信などに伴うイベントを待機し，イベントが発生した場合には，コールバック登録された関数を呼び出す．
- 関数実行機能
  イベントループからの関数呼び出しを受けて，関数情報管理機能から関数のソースコードもしくは中間コードを取得して，そのコードを実行する．
- 関数情報管理機能
  関数のソースコードを管理する．また，関数実行機能からの要求に応じて，関数のソースコードもしくは中間コードを返答する．
- 中間コード管理機能
  関数情報管理機能からの要求に応じて，関数の中間コードを返答する．必要に応じて，関数のソースコードをパーズして中間コードを作成，もしくは中間コードを削除する．

関数実行時，Just in time コンパイラーなどが搭載されていない標準的な JavaScript インタープリターは，そのソースコードをパーズして実行する．これに対して，図 2 の赤

色で示した中間コード管理機能を追加し，ソースコードの代わりに中間コードを使って関数を実行できるようにしたことが，本提案手法のポイントである．

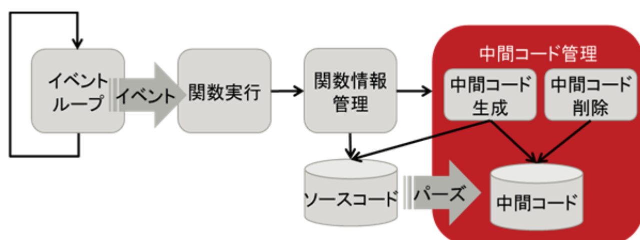

図 2　提案手法の構成図

Figure 2 Diagram of proposal technique.

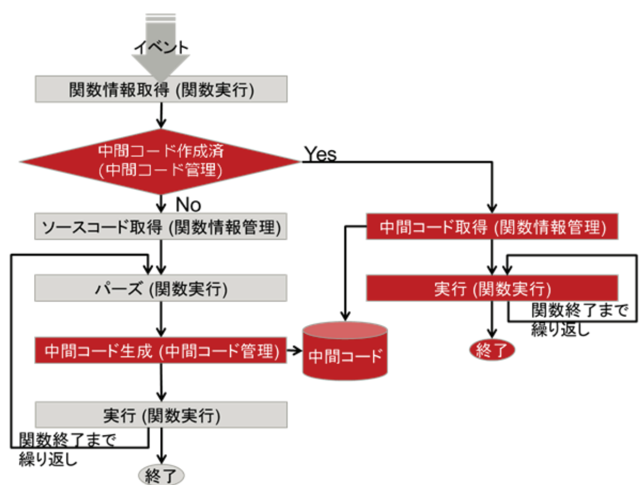

図 3　提案手法のフロー

Figure 3 Flow chart of proposal technique.

図 3 にフローを示す．なお，図内の括弧部分は構成図の項目に一致している．イベントを受けて関数情報を取得するときに，中間コードが作成済みかどうかを判断し，中間コードが作成されていない場合には，ソースコードをもとにパーズ・実行を関数終了まで繰り返す．その際に，合わせて中間コードを作成する．中間コードが作成済みの場合は, 中間コードを取得して, 実行を関数終了まで繰り返す．ソースコードの代わりに中間コードを使うことで，関数部分のソースコードをパーズする処理を省略することができ，その時間を短縮できることになる．

4. 実装

提案手法を実装するにあたり, JavaScript エンジンとして, Espruino[13]を使用することにした．Espruino は 48kbyte の RAM しか持たない低スペックの Espruino ボード上で動作させることを想定しており，小型デバイスでの動作に適している．この Espruino に実装を追加して，本提案手法を実現する．

(1) 実装

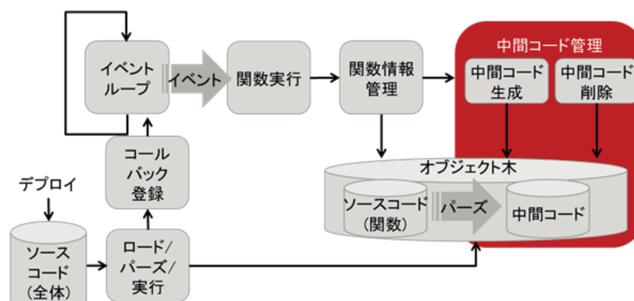

図 4　実装図

Figure 4 Implementation diagram.

図 4 に実装図を示す．Espruino では JavaScript で記述されたテキストファイルであるソースコードをロードした後，そのソースコードを前から順にパーズと実行を繰り返す（ロード/パーズ/実行）．そして，実行により得られたオブジェクト（変数や関数）を木構造（オブジェクト木）として，メモリ上で管理する．このオブジェクト木を使うことで，後からそのオブジェクトを取得してアクセスすることが可能になっている．関数はオブジェクトの一つであり，ソースコードのパーズ時に関数定義があった場合には，関数名と引数，関数部分のソースコードを対応付けて，オブジェクト木に追加して管理する．また，コールバック登録があった場合には，ライブラリを通してイベントと関数を対応付けて記憶する．登録したことにより，イベント発生時に対応付けた関数が呼び出されることになる．関数呼び出し時には，そのオブジェクト木を使って呼び出された関数に対応するソースコードを関数情報管理経由で取得し，その取得した関数部分のソースコードをパーズ・実行する（関数実行）．この関数情報管理部分を拡張して，中間コード管理機能が動作するように実装した．なお，ここで管理する中間コードは，関数のソースコードをステートメントごとにパーズして得られた結果から作成した木構造をリストにしたバイナリデータである．

まず，オブジェクト木を拡張して，関数名とそのソースコードの対応づけに加えて，中間コードへの対応づけを持つことにした．そして，関数情報管理は，関数呼び出し時に関数名でオブジェクト木から関数のソースコードを取得するのに代えて，対応する中間コードが存在する場合は，中間コードを返答する．関数実行により，その中間コードを実行する．中間コードが存在しない場合は，関数のソースコードをパーズ･実行しながら, 中間コード生成を行い，オブジェクト木に関数名と対応づけて記憶する（中間コード管理）．これにより，次回以降の呼び出し時に中間コードを使った実行が可能となり，高速化が実現される．

この方法では，中間コードが作成されるのは，呼び出された関数に限られるが，たとえ限定されたとしても，中間コードを記憶するためにメモリ領域が必要である．したがって，多くの関数があった場合に，小型デバイスではメモ

リの使用量が大きすぎることになりうる．そのため，中間コードを削除する機能を追加した．中間コードによって使用されるメモリ使用量を管理し，一定サイズになった場合に，いずれかの関数の中間コードを削除し，オブジェクト木の対応づけを書き換える．その結果，中間コードを削除された関数が呼び出された場合にも，正常に実行することができる．

(2) **動作フロー**

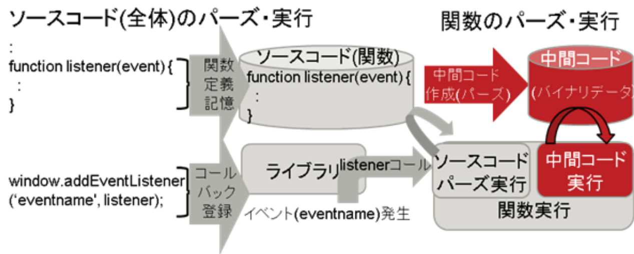

図 5　フロー

Figure 5 Behavior flow.

具体的な流れを図 5 に示す．左のような JavaScript ソースがあった場合，function から始まる関数定義の部分は，抜き出されて管理される．window から始まるライブラリの呼び出しが実行されて，コールバック登録される．そして，対応するイベントが発生した時には，その登録された関数（上記では listener）が呼び出される．そして，オリジナルの Espruino では，listener に対応する関数のソースコードを取得して実行する．本実装では，その代わりに，中間コードが存在する場合は，その中間コードを実行し，存在しない場合は，パーズして中間コードを作成しながら実行する．そこで作られた中間コードは，関数（上記では listener）と対応付けてバイナリデータとして記憶する．

(3) **中間コードとその管理方法**

中間コードについて説明する．まず，JavaScript のソースコードは，一連の表現（expression）をテキスト化したものである．たとえば，図 6 の左に示すシンプルなソースコードを考える．JavaScript エンジンは上記のソースコードを与えられた場合，字句解析・構文解析により，図 6 の右のように表現として解析する．

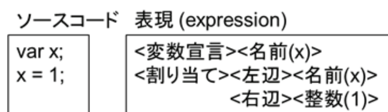

図 6　ソースコードと表現

Figure 6 Source and expression.

JavaScript の 1 行目は変数宣言のステートメントであり，これを 1 つの木構造として表現する．また，2 行目は割り当てのステートメントであり，左辺・右辺両方を解析した上で 1 つの木構造として表現する．そして，これらをリスト化したものを中間コードとしてメモリ上に記憶する．なお，上記はわかりやすさのために言葉で表現しているが，実際にはそれぞれの表現に値を割り当てたバイナリデータである．

次に，実行時に作成されるオブジェクト木について説明する．たとえば，図 7 のソースコードを想定する．

ソースコード

```
var x = 1;
function func1(y) {
  return y + 1;
}
```

図 7　関数ソースコード

Figure 7　Function source code.

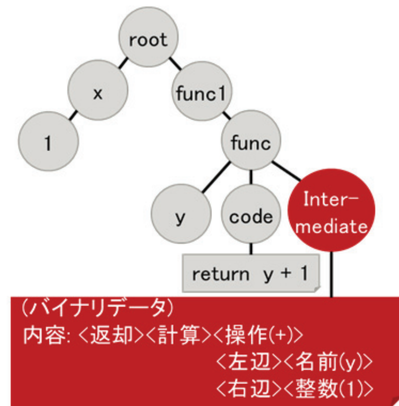

図 8　オブジェクト木の例

Figure 8 A example of object tree.

ソースコードを実行した結果，オブジェクト木に，変数オブジェクト x およびその値である整数 1 を追加し，関数オブジェクト func1 およびその引数 y とソースコード return y + 1;を追加する．その結果として作成されるオブジェクト木のイメージを図 8 に灰色で示す．

Espruino では，関数の呼び出し時に，関数名をもとにこのオブジェクト木を使って関数を検索して，そのソースコードを取得して実行する．これらに対して，関数呼び出し時に，ソースコードを取得する代わりに，中間コードが存在する場合には取得して、その中間コードを実行する変更を加えたことになる．また，中間コードがない場合は，中間コードを作成してオブジェクト木の関数オブジェクトに追加して管理することになる．その部分のイメージを図 8 の赤色で示す．

このように，変数や関数を管理するオブジェクト木で，関数の中間コードについても管理する．

5. 評価

本章では，前章で説明した実装を評価する．評価にあたっては，ハードウェアとして Raspberry Pi Model-B[14]，OS として Raspberry Pi 向けの Linux OS である Raspbian

（Version: January 2014）を使用した．なお，Raspberry Pi はプロセッサ ARM1176JZF-S（周波数 700MHz），メモリ 512MB を搭載したシングルボードコンピュータである．その上で，提案手法を実装した Espruino を動作させた．

(1) シーケンシャル処理での評価

オリジナルの Espruino はインタープリターであり，関数実行においても，関数のソースコードのパーズと実行を順に関数が終了するまで繰り返す．一方で，本提案手法では，中間コードが存在しない場合には関数実行に合わせて中間コードを作成し，中間コードが存在する場合にはソースコードの代わりに中間コードを使用する．そこで，ソースコードをパーズ・実行する時間（オリジナル）と中間コードを作成しながらソースコードをパーズ・実行する時間，中間コードを実行する時間を比較する．中間コードの作成時間や中間コードを使って実行することで省かれるソースコードのパーズ時間は，関数の大きさ，すなわち関数に含まれるステートメント数に応じて変化すると考えられる．そこで，図 9 で示した関数を使い，ステートメント数を横軸として測定し，効果を評価する．

ソースコード
```
function func(param) {
  var x = param;
  x += 10;
  : (横軸にあたるステートメント数個“x += 10;”が続く)
  return x;
}
```

図 9　シーケンシャル処理のソースコード

Figure 9 Source code of sequential process.

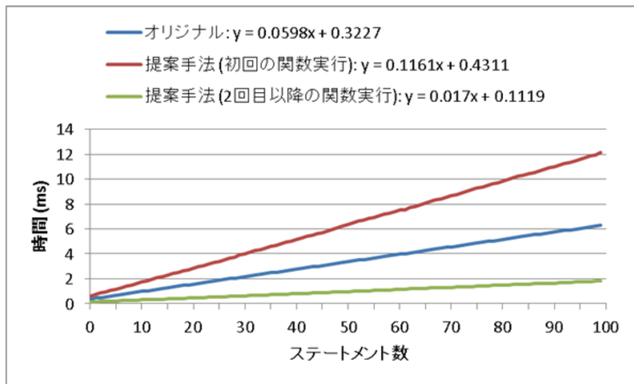

図 10　実行速度（シーケンシャル処理）

Figure 10 Measurement of sequential process.

結果を図 10 に示す．横軸はステートメントの数である．「オリジナル」と記述したグラフは，中間コードを使わずにソースコードからパーズ・実行したときの時間であり，オリジナルの Espruino で関数を実行する時間に当たる．「提案手法（初回の関数実行）」は，ソースコードからパーズ・実行する時間に加えて，パーズ時に中間コードの作成処理を行う時間を含む．これは，提案手法を実装した Espruino で，中間コードが存在しない初回の呼び出し時に関数を実行するときにかかる時間に当たる．「提案手法（2 回目以降の関数実行）」はソースコードの代わりに中間コードを使って関数を実行する時間であり，提案手法を実装した Espruino で，すでに中間コードが作成されている 2 度目以降の呼び出し時に関数を実行するときにかかる時間に当たる．

どのグラフにおいても，ステートメントの数にほぼ比例して実行時間が増加した．この結果は，ステートメントが同じだった場合には，そのステートメントのパーズおよび中間コードの作成には同じ時間がかかるためであり，実行時間は一次近似できると考えられる．そこで，回帰直線を求め，方程式を凡例の部分に示した．これらの結果から，中間コードを作成しながら関数を実行する場合，中間コードを作成しない場合に比べて，およそ 1.94（= 0.1161/0.0598）倍の時間がかかることがわかる．一方で，中間コードを実行する場合は，ソースコードから実行する場合に比べて，0.28（= 0.017/0.0598）倍の時間で実行できることがわかる．したがって，関数が 1 回もしくは 2 回しか呼び出されない場合には，関数の総実行時間はオリジナルのほうが短く，3 回以上呼び出される場合には，提案手法のほうが短い．なお，方程式の定数部分の差は，ソースコードには，可変となっているステートメント（“x += 10;”）以外のステートメントが存在するためと考えられる．

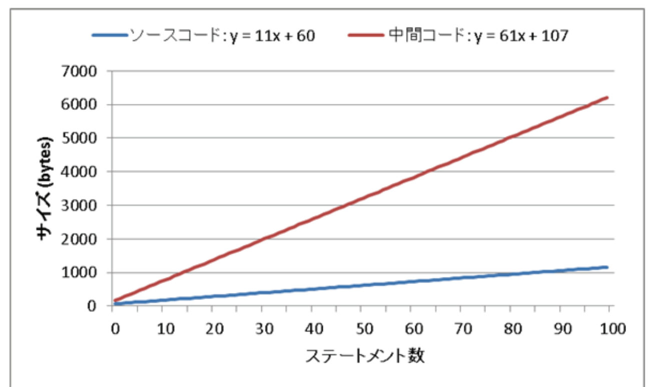

図 11　中間コードサイズ（シーケンシャル処理）

Figure 11 Intermediate Code Size of sequential process.

また，提案手法により作成される中間コードのサイズ（メモリ使用量）を図 11 に示した．「ソースコード」が中間コードのもととなる JavaScript のソースコードのサイズ，「中間コード」が中間コードのサイズである．なお，同じステートメントであれば，同じ中間コードになるため，中間コードのサイズは一次近似できる．方程式を凡例の部分に示した．

メモリ使用量は，ステートメントが 1 つ増えるにつき，61 バイト増加している．元のステートメントは 11 バイト

であり，中間コードはおよそ 5.55（= 61/11）倍のサイズになっている．方程式の定数部分は，可変となっているステートメント（“x += 10;”）以外の部分に当たる．

(2) 繰り返し処理での評価

ソースコードが for や while などの繰り返しを含む場合，オリジナルの Espruino では，繰り返し回数分，同じソースコードのパーズ・実行を繰り返すことになる．本提案手法では，パーズの時間が省かれるため，ステートメント数が少なかったとしても，繰り返し回数に応じて効果が期待できると考えられる．そこで，図 12 に示す for 文を含む場合の繰り返し回数に応じた実行時間を測定し，効果を評価する．

ソースコード
```
function func(param) {
  var x = param;
  for (var i = 0; i < (繰り返し回数);i++) x += 10;
  return x;
}
```

図 12　繰り返し処理のソースコード

Figure 12 Source code of repeat process.

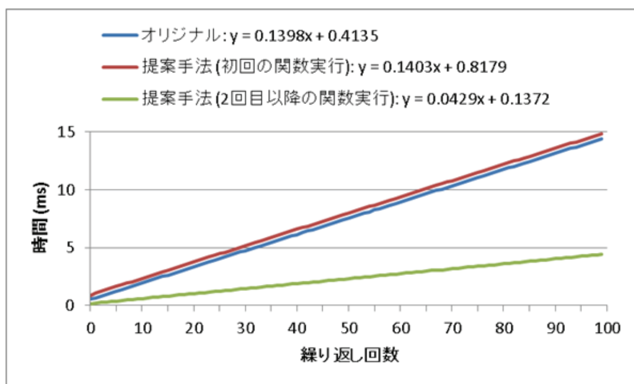

図 13　速度評価（繰り返し処理）

Figure 13 Measurement of loop process.

結果を図 13 に示す．どのグラフにおいても，繰り返し回数にほぼ比例して実行時間が増加した．そこで，回帰直線を求め，それらの方程式を凡例の部分に示した．これらの結果から，中間コードを作成しながら関数を実行する場合，中間コードを作成しない場合に比べて，およそ 1.00（=0.1403/0.1398）倍の時間，すなわちほとんど変わらないこと，つまり，繰り返し回数の増減は中間コードの作成時間に影響しないことがわかる．これは，インタープリターとしての実行では，for 文内を繰り返し回数分パーズ・実行するが，中間コードの作成では，for 文内についても各々のステートメントが中間コード化されるだけであり，パーズ処理は繰り返されないためである．このことは，図 14 に示すように，中間コードのサイズは繰り返し回数によらず 335 バイトで一定であったことからも確かめられる．また，中間コードのサイズが一定であることから，for 文などによる繰り返し回数が多い場合，作成される中間コードのサイズに対して，効果が大きいことがわかる．

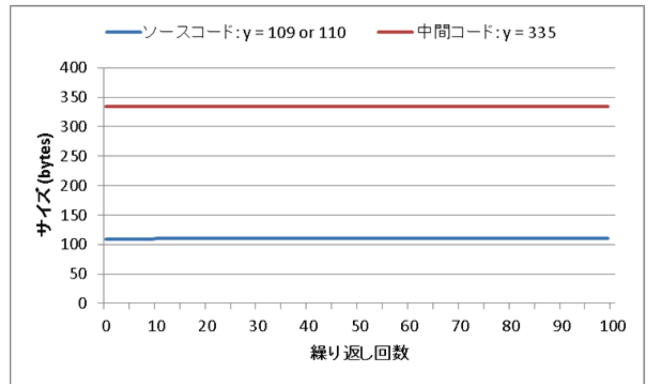

図 14　中間コードサイズ（繰り返し処理）

Figure 14 Intermediate code size of loop process.

また，中間コードを実行する場合は，ソースコードから実行する場合に比べて，0.31（= 0.0429/0.1398）倍の時間で実行できることがわかる．これは，前節のシーケンシャル処理に比べて，高速化の効果が小さい．本提案手法は，ソースコードをパーズ・実行する際のパーズ処理のみに関わるため，実行時間に比べてパーズに時間がかかる場合ほど効果が大きく測定される．for 文内には，前節（図 9）のシーケンシャル処理と同じステートメント（x += 10;）を使っているが，繰り返し処理の実行時には，繰り返しのためのインクリメント処理（i++）および比較処理（i <（繰り返し回数））も合わせて実行される．比較処理の場合，計算処理に比べて，パーズ時間に対する実行時間が長く，効果に差が出たと考えられる．また，これらの結果より，2 回以上呼び出される場合には，本方法が早く実行できることがわかる．

(3) メモリ使用量

小型デバイス上で動作させるには，メモリ使用量が少ないことが望まれる．(1)節および(2)節では，中間コードによるメモリ使用量の増加を測定した．ここでは，JavaScript エンジン自体が必要とするメモリ使用量を比較する．オリジナルの Espruino と提案手法を実装した Espruino のメモリ使用量の比較を表 1 に示す. また, 参考のため, 同 Raspberry Pi 上でビルドした V8 についても示した．なお， JavaScript を実行しない場合、実行に必要な部分の多くがメモリ上に読み込まれないため、単純なソースコード（図 9 で示したソースコードでステートメント数が 0 の場合）を実行して比較した．提案手法を実装した Espruino のメモリ使用量は実装前に比べて，52k bytes の増加に留まっており，小型デバイスを対象とした JavaScript エンジンの範囲に収まると言える．また，Google V8 JavaScript Engine は，提案手法を実装した Espruino と比べてメモリ使用量は 22 倍と非常に大きく, 低スペックのデバイスには適さないと考えられる．

なお，メモリ使用量の測定には，smem ツール[15]を使用した．

表 1 メモリ使用量

Table 1 Memory Usage

JavaScript エンジン	メモリ使用量
オリジナルの Espruino	241K bytes
提案手法を実装した Espruino	293K bytes
Google V8 JavaScript Engine	6,330K bytes

(4) 実行形式との比較による評価

JavaScript は事前のコンパイルが不要でさまざまな環境で実行可能である反面，デバイス向けにコンパイルされた実行形式のバイナリファイルを実行するのに比べて，速度が遅い．ここでは，C 言語で記述してコンパイルした実行形式のバイナリファイルの場合と本方式による中間コードの場合を比較することで，実行時間の違いを確認する．

まずは，(1)節で示したシーケンシャル処理を測定した．使用した JavaScript のソースコードは図 9 に示したものであり，バイナリファイルは JavaScript のソースコードをそのまま C 言語の形式に修正してコンパイルしたものである．結果を図 15 に示す．

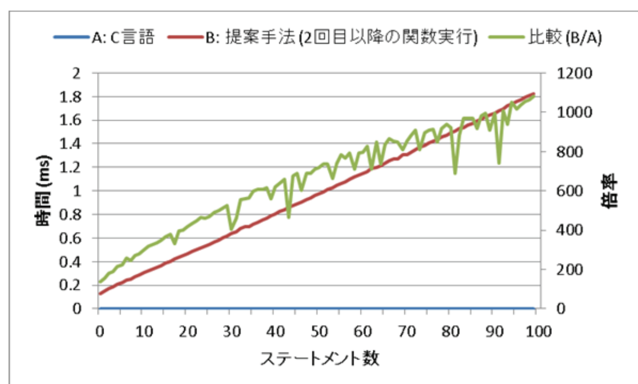

図 15 バイナリファイルとの比較（シーケンシャル処理）

Figure 15 Comparing with binary file.

本提案手法により JavaScript の実行が高速化されたものの，バイナリファイルの実行に比べると，依然として大きな差がある．特に，ステートメント数が増えるほど差が大きくなり，ステートメントが 100 で約 1100 倍程度になっている．バイナリファイルの実行が非常に速いため，ステートメント数が小さい時には特に，関数内の実行時間に対して，測定対象である関数の呼び出し処理や時間の測定に必要な処理などのオーバーヘッドの割合が大きくなるため，ステートメント数による違いが現れていると考えられる．

次に，C 言語で扱う場合に実行に時間がかかる例として，文字列の連結処理の速度比較を行った．JavaScript のソースコードを図 16 に示す．

結果を図 17 に示す．繰り返し回数 100 回で 35 倍程度の差であり，シーケンシャル処理と比べると，バイナリファイルとの差は小さくなっている．シーケンシャル処理の場合と異なり，繰り返し回数による倍率の差は小さく 35 倍～83 倍の間に収まっている。これは、繰り返す部分の実行に時間がかかるため、他の部分のオーバーヘッドの割合が小さいためと考えられる。なお，C 言語でも同様の処理を行う関数を作成して比較したが，文字列の連結の場合，C 言語で記述すると，メモリ領域を確保（malloc）して，連結する 2 文字列をコピー（strcpy/strcat）することになる．それと比べて，JavaScript ではメモリ確保や開放がないため，差が小さくなっていると思われる．このことから，JavaScript が得意とする処理では，バイナリファイルとの差は縮まることがわかる．

ソースコード

```
function func() {
  var x = 'a';
  var y = '';
  for (var i = 0; i < (繰り返し回数);i++) y = y + x;
  return 0;
}
```

図 16 繰り返し（文字列連結）処理のソースコード

Figure 16 Source code of repeat process

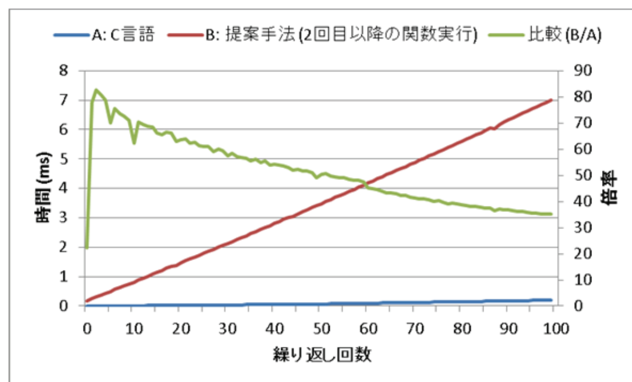

図 17 ネイティブとの比較（文字連結）

Figure 17 Comparing with Native Code

(5) 考察

これらの結果から，中間コードを使う本提案手法により，(1)節で示したシーケンシャル処理で 0.28 倍，(2)節で示した繰り返し処理で 0.31 倍の時間で関数を実行することが可能となった．中間コードの作成処理については，(1)節で示したシーケンシャル処理で 1.94 倍，(2)節で示した繰り返し処理で 1.00 倍の時間で実行できたことから，イベント処理のように複数回呼ばれる関数に本手法を提供することは有効であると言える．

また，本手法の中間コードが必要とする記憶領域は，関数内のステートメント数に依存する．そのため，ステートメント数は大きくないものの for 文などによる繰り返し処

理が多い関数に本手法を適用した場合，使用メモリをあまり増やさずに大きな効果が得られる．したがって，使用メモリに制限がある場合には，ソースコードの大きさと実際の処理量をもとに中間コード生成・中間コード削除処理を行うことが適切である．また，提案手法により，バイナリファイルの実行速度に近づいているものの，まだ大きな差があることから，高速化の余地があると考えられる．

6. 結論

小型デバイス上で動作するイベントドリブンモデルを想定し，その上で動作させる JavaScript エンジンの高速化手法を提案した．また，提案手法を，低スペックのデバイス上で動作可能な JavaScript エンジンである Espruino に実装し，Raspberry Pi 上で動作させた．その上で，ステートメント数や繰り返し回数を変えた JavaScript ソースコードを実行して，効果を測定・評価した．その結果，関数の実行時間が本手法を適用しない場合に比べて、0.28〜0.31 倍に抑えられる、つまり 3.23〜3.57 倍に高速化することを示した．したがって，イベント処理のような複数回呼ばれる関数を持つ JavaScript を実行する場合，本手法による改善効果は十分高いと言えるであろう．

Internet of Things/Web of Things の実現が進むにつれて，モノの Web 化でサービスが広がり，センサー等のイベントに基づき動作する小型デバイスが重要になると思われる．このようなデバイスでは，メモリ使用量を考慮した上で，イベントに応じて呼び出される関数を高速に実行することが重要である．そのため，本提案手法のように，メモリ使用量を大きく増加させずに高速化を達成する手法が有用になると考えられる．

参考文献

1) W3C Community and Business Groups: Web of Things Community Group http://www.w3.org/community/wot/
2) Joyent Inc.: Node.js http://nodejs.org/
3) Google Inc.: V8 JavaScript Engine https://code.google.com/p/v8/
4) Technical Machine: Tessel Hardware development for software developers https://tessel.io/
5) Stein, G.: Inside Tessel, The JavaScript Microcontroller That Is Changing Everything http://www.fastcolabs.com/3021917/open-company/inside-tessel-the-javascript-microcontroller-that-is-changing-everything (2013).
6) WebKit Org.: The WebKit Open Source Project http://www.webkit.org/projects/javascript/
7) Mozilla Developer Network: SpiderMonkey https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey
8) Ha, Jungwoo, et al.: A concurrent trace-based just-in-time compiler for single-threaded JavaScript, PESPMA 2009, pp.47-pp.54 (2009).
9) Lee, Seong-Won, et al.: Code size and performance optimization for mobile JavaScript just-in-time compiler, Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture. ACM, pp.6-pp.12 (2010).
10) Jeon, Sanghoon, and Jaeyoung Choi: Reuse of JIT compiled code in JavaScript engine, Proceedings of the 27th Annual ACM Symposium on Applied Computing. ACM, pp.1840-pp.1842 (2012).
11) Aho, Eero, et al.: Towards real-time applications in mobile web browsers, Embedded Systems for Real-time Multimedia (ESTIMedia), 2012 IEEE 10th Symposium on. IEEE, pp.57-pp.66 (2012).
12) Pitambare, Uday, et al.: Just-in-time Acceleration of JavaScript, Technical Report TR706, School of Informatics and Computing, Indiana University, Bloomington, Indiana, USA, February 2013 (2013).
13) Pur3 Ltd.: Espruino JavaScript for Microcontrollers http://www.espruino.com/
14) Raspberry Pi Foundation: Raspberry Pi http://www.raspberrypi.org/
15) Selenic consulting: smem memory reporting tool http://www.selenic.com/smem/