東京大学
情報理工学系研究科 創造情報学専攻
修士論文

保守的に挿入しても速度低下が小さい JavaScript 向け await 互換演算子のコード変換による実装

Implementation of await-compatible operator with low overhead when used conservatively for JavaScript

川向 聡
Satoshi Kawamukai

指導教員　千葉 滋 教授

2024 年 1 月

概要

JavaScript のプログラムにはしばしば非同期処理が登場する．JavaScript の `await` 演算子は本来, 非同期関数の終了を待つための演算子だが, どの関数が非同期関数であるかは自明ではない．`await` の書き忘れによるバグを避けるため, すべての関数呼び出しに `await` を付与することが有効と考えられるが, 同期関数の呼び出しに本来不要であるはずの `await` を付与すると, 実行速度が大きく低下する．

本研究では, 同期関数の呼び出しに付与した場合でもオーバヘッドの小さい `await` 互換の演算子 `await?`を導入し, これを JavaScript 上で実現するためのコード変換を提案する．我々が提案する `await?`では, `Promise` オブジェクトを処理するスローパスと, 高速に動作するファストパスを設け, どちらを実行するかをプログラムの実行時に選択する．ファストパスは `await?`が付与された式が `Promise` オブジェクト以外を返す場合に実行され, オーバヘッドを抑える．また, コード変換器と非同期処理を含むベンチマークを開発し, `await?`の性能を評価する実験を行った．実験により, `Promise` オブジェクト以外を返す式に付与された場合は実行時間が概ね短縮され, `Promise` オブジェクトを返す式に書かれた場合には同等の性能を示すことを確認した．

Abstract

Asynchronous processing often appears in JavaScript programs. The `await` operator in JavaScript is an operator to wait for the end of an asynchronous function, but it is not obvious which function is an asynchronous function. To avoid bugs caused by forgetting to write `await`, it is considered effective to add `await` to all function calls. However, adding the original `await` to calls of synchronous functions, which are not necessary, causes a significant decrease in execution speed.

In this research, we introduce an `await`-compatible operator `await?` which has a small overhead if it is added to synchronous function calls, and propose a code translation to realize `await?` on JavaScript program. `await?` has a "slow path" for processing `Promise` objects and a "fast path" for fast execution. The choice of which one to execute is made at run time. A "fast path" is executed when an expression with `await` returns a non-`Promise` object, thus reducing the overhead. Also we developed a code translator and benchmarks including asynchronous processing and evaluated the performance of `await?`. We have confirmed that the execution time is generally reduced when it is added to an expression that returns a non-`Promise` object, and that the performance is equivalent when it is added to an expression that returns a `Promise` object.

目次

第 1 章

はじめに

JavaScript と非同期処理は不可分の関係にある．JavaScript は Web アプリケーションを記述する際によく用いられており, ほとんどのブラウザがその実行エンジンを備えている．このような利用シーンでは, HTTP リクエストのような結果の返却に時間がかかる処理がしばしば登場する．処理の終了を待つ間, ブラウザのレンダリングといった他の処理が停止するのを避けるため, 時間のかかる処理は非同期に実行される必要がある．

JavaScript では, 非同期処理を扱うために Promise というオブジェクトが導入されている．また, Promise オブジェクトを明示的に操作するよりも簡便に非同期処理を記述する方法として async/await 構文が利用できる．Promise オブジェクトは非同期処理が完了したかどうかの状態とその結果を持つオブジェクトであり, await は Promise オブジェクトに対応した非同期処理の完了を待つ演算子である．

標準の await は, Promise オブジェクトを返さない式に付与したとき, オーバヘッドが大きいという問題点がある．async/await 構文を用いて非同期処理を記述するとき, Promise オブジェクトを返す関数の呼び出しに対して await を書き忘れることによるバグがしばしば生じる．この対策として, Promise オブジェクトを返すものとそうでないものを区別せず, すべての関数呼び出しに保守的に await を付与することが有効と考えられるが, 標準の await を利用した場合, そのオーバヘッドによりプログラムの実行速度が大きく低下してしまう．

本論文では, この問題を解決するために Promise オブジェクト以外を返す式に付与してもオーバヘッドの小さい await?という演算子を新たに設け, await?を JavaScript 上で実現するためのコード変換について提案する．提案する await?は, コード変換によってスローパスとファストパスという 2 つのパスをプログラムに構築し, Promise オブジェクト以外を返す式に付与されていた場合は実行時間の短いファストパスへ分岐することでオーバヘッドの削減を実現する．加えて, 提案したコード変換を行うプリプロセッサを実際に開発した．

await?の性能を評価するため, ベンチマークの実行時間を標準の await を利用した場合と比較する実験を行った．await?を一部の関数呼び出しに付与したのちにコード変換を行ったベンチマークと, await を一部の関数呼び出しに付与したベンチマークを用意して実行時間を計測し比較を行った．また, 提案したコード変換の手法を用いたとき, コードサイズがどの程度増加するかについても測定した．

本研究の貢献は大きく以下の 3 つである.

1. 保守的な利用を想定し, `Promise` オブジェクト以外を返す式に付与した場合の速度低下が標準の `await` より小さい演算子 `await?`を提案した.
2. `await?`を JavaScript 上で実現するためのコード変換を提案し, `await?`を含むプログラムを純粋な JavaScript コードに変換する変換器を開発した.
3. `await?`の性能評価にあたって, Marr らの Cross-Language Compiler Benchmarking [1] に含まれるベンチマークを元に, 非同期処理を含むベンチマークを作成した.

本論文の構成について述べる. 第 2 章では, 本研究の前提知識について述べ, 関連研究を紹介する. 第 3 章では, 新たな演算子 `await?`とそれを JavaScipt 上で実現するためのコード変換の手法について詳細に述べる. 第 4 章では, `await?`の性能を評価するために作成したベンチマークの詳細, 及びコード変換のバリエーションについて説明したのち, 実験結果について考察する. 第 5 章では, 本研究のまとめと今後の課題について述べる.

第 2 章

JavaScript における非同期処理

プログラムにおいて, タスクの完了を待たず他のタスクの処理を始める処理の方式を非同期処理という. 非同期処理は通信が絡むアプリケーションに広く利用されており, ユーザへ迅速な応答を提供しながら別の処理を進めるための有用な手段となっている. しかし, 非同期処理を含むプログラムは, 処理の流れがソースコード上で登場する順番と必ずしも一致しないといった理由から, その全容をプログラマが把握するのが困難になり, 様々なバグを招くおそれがある. このような問題に対処するため, 非同期処理を含むプログラムの処理の流れを解析する手法や, 間違った非同期処理の実装方法をプログラマに伝えるツールの開発がなされている.

本章では, 本研究の前提知識となる, 同期処理と非同期処理について, 及び JavaScript で非同期処理を扱う方法について説明する. そして, そのうちのひとつである `async`/`await` を用いた方法において, バグを抑制する際に障壁となる実行時間に関する問題点について述べる. また, JavaScript における非同期処理関連の問題の解決を試みた関連研究についても紹介する.

2.1 前提知識

2.1.1 同期処理と非同期処理

プログラムにおいて, ソースコードに書かれたタスクを上から順番に処理していく方式を同期処理という. この場合, タスクがどんなに時間のかかるものであったとしても, そのタスクが完了するまで以降のタスクの実行はブロックされる. そのような処理がブラウザのメインスレッドで実行されると, 処理が終わるまでレンダリングが行われず画面が固まるといったことが起きる. 一方で, タスクの完了を待たずに他のタスクの処理を進める処理の方式を非同期処理という. ブラウザ上で実行されるプログラムでは, データベースへのアクセスといった完了まで時間のかかる処理を非同期に実行することで, 画面が固まるといった問題を避けられる.

JavaScript の `setTimeout()` メソッドを利用すると, 指定した関数を一定時間経過したあとに実行するようにタイマーを設定することができる. プログラム 2.2 では, 12 行目で関数 `task2()` を 1 秒後に実行するよう設定している. ソースコード上では `task1()`, `task2()`, `task3()` の順で並んでいるものの, 実際の実行順は `task1()`, `task3()`, `task2()` となる.

```
function task1() {
    console.log("1")
}
function task2() {
    let sum = 0
    for (let i = 0; i < 1e9; i++) {
        sum += i
    }
    console.log(sum)
}
function task3() {
    console.log("3")
}
task1()
task2() // heavy
task3()
// task2 が終わるまで task3 は実行されない
```

プログラム 2.1: 同期処理の例

```
function task1() {
    console.log("1")
}
function task2() {
    console.log("2")
}
function task3() {
    console.log("3")
}

task1()
setTimeout(task2, 1000) // 1000 ミリ秒後に task2 を実行
task3()
// 1 3 2 の順番で出力される
```

プログラム 2.2: 非同期処理の例

```
1 const fs = require("fs")
2
3 fs.readFile("foo.txt", (error, data) => {
4     if (error) {
5         console.error(error.message)
6         process.exit(1)
7     }
8     // 読み込んだデータを使う処理
9     doSomething(data)
10 })
```

プログラム 2.3: コールバック関数を利用した追加の処理の登録

2.1.2 JavaScript で非同期処理を扱う方法

非同期処理を含むプログラムには, 非同期に実行される処理が終わったあと, その結果を使って別の処理を行いたいというシーンが登場する. また, 非同期処理の最中に発生した例外を扱うためには特別な対応が必要となる. 本節では, JavaScript で非同期処理を扱う方法として, コールバック関数のみを用いる方法, `Promise` オブジェクトを利用する方法, `async`/`await` 構文を利用する方法について述べる.

コールバック関数のみを用いる方法

非同期に実行された処理が, ソースコード上のどの位置で完了しているかを判断することはできないが, コールバック関数を利用することで, その結果を使って行いたい処理を登録することができる. プログラム 2.3 では, Node.js[*1]が提供する `fs` モジュールの `readFile()` メソッドを利用してファイルを読みとっている. ファイルの読み取りは時間のかかる処理であり, `fs.readFile()` メソッドは非同期に実行されるため, ファイルから読み込んだデータを使って行う処理はコールバック関数として登録する. プログラム 2.4 に示したように, 複数のファイルから読み込んだデータを使って処理を行いたい場合は, コールバック関数を何度も記述する必要がある. このようにネストが深くなる問題はコールバック地獄と呼ばれ, ソースコードの可読性を下げる要因となる.

JavaScript では `try...catch` 構文を利用することで例外に対応することができるが, `try` ブロック内の非同期処理において発生した例外は補足することができない. プログラム 2.5 では, 8 行目が実行された後に `setTimeout()` メソッドに登録したコールバック関数が実行され, `try` ブロックの範囲外で例外が発生するため, `catch` ブロックは実行されない. コールバック関数の中で `try...catch` 構文を使うと例外を補足することができるが, コールバック関数の外へ例外の発生を伝えることはできない.

[*1] `https://nodejs.org/en`

```
const fs = require("fs")

fs.readFile("foo.txt", (error, data1) => {
    if (error) {
        console.error(error.message)
        process.exit(1)
    }
    fs.readFile("bar.txt", (error, data2) => {
        if (error) {
            console.error(error.message)
            process.exit(1)
        }
        fs.readFile("baz.txt", (error, data3) => {
            if (error) {
                console.error(error.message)
                process.exit(1)
            }
            // 読み込んだデータを使う処理
            doSomething(data1, data2, data3)
        })
    })
})
```

プログラム 2.4: コールバック地獄

```
try {
    setTimeout(() => { throw new Error("error") }, 1000)
} catch (e) {
    console.log(e) // 実行されない
}
console.log("end")
```

プログラム 2.5: 補足できない例外

```
setTimeout(() => {
    try {
        throw new Error("error")
    } catch (e) {
        console.log(e) // 実行される
    }
}, 1000)
```

プログラム 2.6: 補足できる例外

```
const fs = require("fs")

const p1 = new Promise((resolve, reject) => {
    fs.readFile("foo.txt", (error, data) => {
        if (error) {
            reject(error) // 読み取り失敗
        }
        resolve(data.toString()) // 読み取り成功
    })
})
```

プログラム 2.7: Promise オブジェクトの作成

```
p1.then(
    (value) => {
        console.log(value)
    },
    (error) => {
        console.error(error.message)
    }
)
```

プログラム 2.8: then() メソッドによるコールバック関数の登録

Promise オブジェクトを利用する方法

より簡単に非同期処理を記述する方法として, ECMAScript2015[*2]で導入された Promise[2] というオブジェクトを用いる方法がある．Promise オブジェクトは非同期処理の結果およびその値を表し, 初期状態を表す”pending”, 処理が成功したことを表す”fulfilled”, 処理が失敗したことを表す”rejected”という 3 つの状態を持つ．pending 状態にある Promise オブジェクトは, 最終的に結果の値を持つ fulfilled 状態またはエラーの理由を持つ rejected 状態に遷移する．fullfilled 状態または rejected 状態に遷移したとき, Promise.prototype.then() メソッドや Promise.prototype.catch() メソッドなどによって登録したコールバック関数が呼び出される．また, 既に fullfilled 状態または rejected 状態となった Promise オブジェクトにコールバック関数を登録した場合も, 対応したコールバック関数が呼び出される．

Promise オブジェクトを生成する際には, コンストラクタには executor() と呼ばれる関数を渡す必要がある．この executor() は, 非同期処理を行うことが期待される．executor() は引数が 1 つの関数 resolve(), reject() を引数にとる．resolve() は処理の成功を伝えるために executor() によって実行される関数であり, 処理の結果として得られた値を引数にと

[*2] https://262.ecma-international.org/6.0/

```
function readDB(url) {
    return connectDB(url)
        .then((conn) => {
            return read(conn)
        })
        .then((content) => {
            return parseResult(content)
        })
        .catch((error) => {
            console.error(error.message)
        })
}
```

プログラム 2.9: Promise チェーン

る．reject() は処理の失敗を伝えるために executor() によって実行される関数であり, エラーの理由 (通常は Error オブジェクト) を引数にとる．プログラム 2.7 では, executor() の中でファイル読み取りを行い, 読み取りに失敗した場合には Error オブジェクト error を引数にして reject() を呼び出し, 成功した場合には結果 data を引数にして resolve() を呼び出している．

then() メソッドを用いると, 処理の成功と失敗それぞれに対してコールバック関数を登録できる．catch() メソッドを用いると, 処理の失敗に対してコールバック関数を登録できる．プログラム 2.8 では, プログラム 2.7 で作成した Promise オブジェクト p1 に対して then() メソッドを呼び出し, p1 が fullfilled 状態, rejected 状態となった際に呼び出すコールバック関数をそれぞれ登録している．Promise オブジェクトの内部で例外が throw されると, その Promise オブジェクトは rejected 状態となり, then() メソッドあるいは catch() メソッドで登録した処理失敗時のコールバック関数が呼び出される．

then() メソッドや catch() メソッドは, 処理の結果を持ち fullfilled 状態の Promise オブジェクトを返すため, 連続して then() メソッドを呼び出すことで更なる処理を追加することが可能である．これは Promise チェーンと呼ばれる．Promise チェーンを用いると, ネストを深くすることなく非同期処理のあとに行う処理を連鎖させることができる．Promise チェーンのどこかで Promise オブジェクトが rejected 状態になったときは, 途中にある処理をスキップして, 最も近い処理失敗時のコールバック関数が呼び出される．そのため, Promise チェーンの最後で catch() メソッドを呼び出し, 道中で発生しうるエラーの処理をまとめて行う場合が多い．プログラム 2.9 に Promise チェーンの例を示した．関数 connectDB() から返却された Promise オブジェクトに対して then() メソッドを呼び出し, then() メソッドから返却される Promise オブジェクトに対して更に then() メソッドを呼び出すことで, 非同期処理の順序を制御している．また, Promise チェーンの最後で catch() メソッドを呼び出し, 途中でエラーが発生した場合の処理を記述している．

```
async function readDB(url) {
    try {
        let conn = await connectDB(url)
        let content = await read(conn)
        return parseResult(content)
    } catch (error) {
        console.error(error.message)
    }
}
```

プログラム 2.10: async/await 構文を使ったコード

async/await 構文を利用する方法

async/await は ECMAScript2017[*3]で導入された Promise オブジェクトを簡単に扱うための構文である．関数宣言文に async キーワードを付与することで, 関数内部で await という演算子が利用できるようになる．await を Promise オブジェクトを返す式に付与することで, その Promise オブジェクトが fullfilled 状態あるいは rejected 状態になるまで, 以降の処理を行わず待機することができる．await が付与された式が複数ある場合は, その都度処理の完了を待つため, Promise チェーンを明示的に構築することなく実行順序を制御することができる．

async が付与された関数は, 最終的に Promise オブジェクトを返却する．関数が Promise オブジェクトを返却する場合はその Promise オブジェクトを, Promise オブジェクト以外を返却する場合はその値で fullfilled 状態となった Promise オブジェクトを返却する．例外が発生した場合は, エラーの情報を持ち rejected 状態となった Promise オブジェクトを返却する．

Promise チェーンを用いて記述したプログラム 2.9 を async/await 構文を利用して書き換えるとプログラム 2.10 が得られる． Promise オブジェクトを返却する関数である connectDB() と read() の呼び出し式に await を付与することで, それぞれの処理の完了を待ってから次の処理を行うことができる． async を付与した関数の内部では既存の try...catch 構文を利用することができ, try ブロックの中で発生した例外を補足して catch ブロック内でエラー処理を行うことができる．

以降では, async の付与された関数を含め Promise オブジェクトを返す関数を非同期関数, Promise オブジェクト以外を返却する関数を同期関数と呼ぶ．

[*3] https://262.ecma-international.org/8.0/

```
async function forgotten_await() {
  const { request } = require("urllib")
  const { data, res } = request("https://example.com/")
  console.log(res.statusCode, data.length)
}
// TypeError: Cannot read properties of undefined
// (reading 'statusCode')
```

プログラム 2.11: await の書き忘れによるバグを含む関数と呼び出した際のエラーメッセージ

2.2　await のオーバヘッド

我々は, 標準の await を Promise オブジェクト以外を返却する式に付与した場合に, 大きなオーバヘッドが生じることを実験で確認した. 本節では, 非同期処理に関連するバグを抑制する方法として await の保守的な付与を想定した際, このオーバヘッドがその障壁となることについて詳しく述べる.

async/await 構文を利用したプログラムにおけるバグの典型的な原因として, await を書き忘れることが挙げられる. 同期関数と非同期関数をひと目で判別することは容易ではなく, 結果としてプログラマはしばしば await を書き忘れる. プログラム 2.11 に await の書き忘れによるバグを含むプログラムの例を示した. プログラム 2.11 では, HTTP リクエストのレスポンスを受け取って, ステータスコードを返却する関数を定義している. しかし, Promise オブジェクトを返却する request() メソッドの呼び出しの前に await を書き忘れてしまっている. await を書き忘れたことで, request メソッドが返却した Promise オブジェクトは, 変数 data, res に分割代入される. 結果として res には値が代入されず, undefined となった res の存在しないプロパティを読み取ろうとしてエラーが発生する. このように, await の書き忘れによるバグは, エラーメッセージを見てもその原因に気づきにくいことがしばしばある. また, ESLint[*4]などの静的解析ツールを利用することによって await の書き忘れをある程度発見することができるが, 呼び出される関数の本体を静的に決定できる場合など, 発見可能な場合は限定されている.

async を付与した関数の中では, Promise オブジェクト以外を返す式にも await を付与することができる. この場合, await は Promise オブジェクト以外の値で fullfilled された Promise オブジェクトを生成し, Promise オブジェクトを返す式に付与された場合と同じように動作するため, await を付与した場合でも付与しなかった場合と同じ結果が得られる. この性質を利用して, ある関数の内部のすべての関数呼び出しに対し, 同期関数と非同期関数の呼び出しを区別せず保守的に await を付与することで, await を書き忘れることによるバグへの対処が可能であると考えられる.

[*4] https://eslint.org/

表 2.1: await を付与することによる実行時間の増加

ベンチマーク	実行時間 [ms]	await 付与後の実行時間 [ms]	実行時間の比
Bounce	25.0	546.7	2190%
List	44.3	510.4	1150%
Storage	77.3	713.2	922%
Towers	37.9	983.1	2590%
DeltaBlue	19.2	318.1	1650%
Richards	93.9	3183	**3390%**
Json	69.6	636.1	914%

しかし, バグの抑制のためにすべての関数呼び出しに付与するといった使い方を考えたとき, 標準の await では同期関数の呼び出しに付与した場合の実行時オーバヘッドが大きい. 実験により, 同期関数の呼び出し式に await を付与すると, プログラムの実行時間が大幅に増加することを確認した. 実験の結果を表 2.1 に示す. 実験には Marr らの Cross-Language Compiler Benchmarking [1] に含まれるベンチマークを用い, 元のベンチマークと, 関数宣言文に async を付与し内部の関数呼び出しに await を付与する変更を加えたベンチマークの実行時間を計測した. await を付与したことにより, ベンチマークの実行時間は最大で約 3400% にまで増加した. 最も実行時間の増加が小さいものでも 900% 程度の増加がみられ, 標準の await を保守的に付与する方法は実用的とはいえないことがわかる.

await を書き忘れるバグを抑制する方法として await の保守的な付与を考えた際, 同期関数の呼び出しに標準に await を付与することによるオーバヘッドが大きく, 元のプログラムの性能を大きく損なう. 本来, 同期関数の呼び出しに await を付与する必要はなく, そのような await の利用方法が想定されていないものと思われる. 現状の await の実装のままでは, await の利用方法に制限が課されてしまっているといえる.

2.3　関連研究

JavaScript における非同期処理の扱いの難しさは過去にも指摘がなされており [3, 4], 非同期処理に由来するバグへの対処を試みた研究が提案されている．例えば, 静的解析などによりバグを事前に発見することを目標にしたもの [5, 6, 7] や, 解析結果を視覚化しプログラマがバグを生み出さないようすることを目標にしたもの [8, 4, 9] がある．我々のアプローチは, バグを発見するのではなく抑制するために `await` を保守的に付与することを想定し, 実行速度の低下が起きにくい新たな演算子を提案する点が新しい．

Sotiropoulos らは, JavaScript の非同期処理の静的解析のために非同期処理の実行順序を解析する手法を提案している [5]．Sotiropoulos らは, まず Guha ら [10] が提案した JavaScript の計算モデルに非同期処理に関連する部分を追加した．そして, 非同期処理に関連するコールバック関数の実行順序を決定するため, その依存関係を有向非巡回グラフとして表現する手法を提案した．中規模のベンチマークを用いた性能評価実験では, プログラム中の非同期処理同士の実行順序を平均で 79% 決定することができることを確認している．この手法では, `async`/`await` が計算モデルに組み込まれておらず, また静的解析ツールの開発にまでは至っていない．

Rau らは, `Promise` を利用するプログラミングのために線形型システム [11] を組み込んだ新たな言語を考案した [6]．線形型の変数は, 必ず 1 回利用され, 2 回以上利用されないことが保証される．Rau らはケーススタディとして, `Promise` オブジェクトの操作に関連するバグを含む JavaScript のプログラムを提案した言語に変換し, 頻出するバグが発見できることを示している．しかし, 新たに提案した言語に存在しない例外処理などの機能を含むプログラムは変換することができず, また `async`/`await` におけるバグについては扱われていない．

Turcotte らは, JavaScript の非同期処理に関するアンチパターンを 8 つ挙げ, これを発見してユーザに提示するツールを提案した [9]．アンチパターンの例としては, 内部に `await` を含まない `async` の付与された関数や, `await` の付与された式を返却している `return` 文などが挙げられており, 無駄な要素を省くことで性能向上が見込める．Turcotte らは GitHub リポジトリからアンチパターンのインスタンスを抽出し, 性能向上こそ限定的であるものの, アンチパターンの大半が簡単に修正できることを確認している．また, プログラマの非同期処理に対する理解度の向上を目指し, 静的解析及び動的解析で発見したアンチパターン, 動的解析で決定した各 `Promise` の生存期間や依存関係などを視覚化するツールを作成している．

第 3 章

await 互換演算子の提案

本章では, 2 章で述べた標準の await 演算子の問題点を解決するため, await と互換性があり, 保守的に付与した場合でも速度低下の小さい演算子 await?を新たに提案する．そして, await?を JavaScript 上で実現するためのコード変換の方法について説明したのち, 実際に開発したコード変換器について述べる．標準の await には, Promise オブジェクト以外を返す式に付与された場合に大きな速度低下が生じるという問題点があった．await?は標準の await と同じ動作をするが, Promise オブジェクト以外を返す式に付与された場合の速度低下が小さい.

3.1 概要

提案する演算子 await?の特徴は, ファストパスとスローパスという 2 つのパスを予め用意しておき, 実行時に適切なパスを選んで処理を分岐させることにある．await?が Promise オブジェクト以外を返す式に付与されていた場合は, 実行時間の短いファストパスへ分岐することで速度低下を抑えることができる．ファストパスは式が Promise オブジェクト以外を返す前提で最適化されており, 標準の await 相当の処理を省略するため実行時間が短い．一方, スローパスは式が Promise オブジェクトを返す前提で, 標準の await 相当の処理を行う．スローパスでは Promise オブジェクトを処理するが, 標準の await を利用した場合と比較して実行時間に大きなオーバーヘッドが生じないことが期待される．適切なパスへ分岐するために, 実行時に await?が付与されていた式の値を検査し, Promise オブジェクトがそうでないかを確認する.

await?はソースコード変換によって JavaScript 上で実現される．図 3.1 に, await?を含む JavaScript のソースコードを実行するまでの流れを示した．プログラマが記述した await?を含むソースコードは, 我々が開発したコード変換器により, await?の挙動を JavaScript の機能のみを使って実現した純粋な JavaScript コードに変換され, 既存の JavaScript エンジンで実行される.

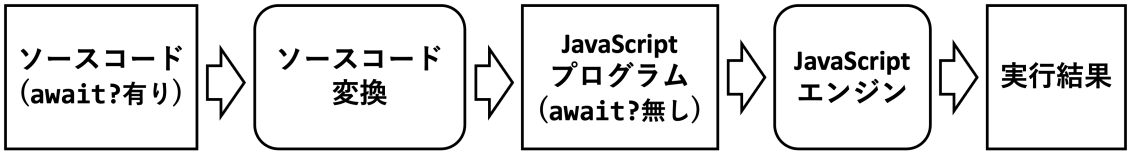


図 3.1: await?演算子を含むソースコードを実行するまでの流れ

```
1 async function sample(url) {
2     let conn = await? connectDB(url)
3     let content = await? read(conn)
4     return content
5 }
```

プログラム 3.1: コード変換の対象となる await?を含むプログラム

3.2　コード変換の流れ

本節では, await?を JavaScript 上で実現するためのコード変換について述べる. なお, 変換は関数単位で実行される. プログラムの実行時に適切なパスを選ぶために, await?が付与された式の返す値が Promise オブジェクトかそれ以外かでスローパスとファストパスに分岐する if 文を挿入する. スローパスには Promise オブジェクトの then() メソッドに await?式以降の継続を渡す処理を記述する. 一方, ファストパスには await?式以降の継続をそのまま実行する処理を記述する. 継続を切り出す際には, 継続渡しスタイル（Continuation Passing Style, CPS）[12] への変換 (以降は CPS 変換と呼ぶ.) を行う. await?を for 文や while 文の中で利用した場合, この CPS 変換が複雑になり, かえって実行速度が低下するため, そのような場合は単に await?を標準の await で置き換えて変換を終了する.

コード変換全体の手順は以下の通りである.

1. ループの中に await?が含まれる場合, 関数内のすべての await?を標準の await で置き換え, 変換を終了する.
2. 関数宣言文から async キーワードを取り除く.
3. 実行順で最初に登場する await?式を変換の対象に選び, 選択した await?式以降の継続を切り出し関数化する.
4. await?が付与された式の結果を変数に束縛する代入文を挿入する.
5. await?の対象である式の結果が Promise オブジェクトかどうかによって分岐する if 文を挿入する.
6. スローパスとファストパスを構築する.
7. 手順 3 で切り出した継続が await?を含む限り, 再帰的に処理する.

```
1 async function unsuitable_sample(n) {
2     let r = 0
3     for (let i = 0; i < n; i++) {
4         r += await? getTemperature(i)
5     }
6     return r / n
7 }
```

プログラム 3.2: コード変換に適さない関数の例

```
1 function sample(url) {
2     let conn = await? connectDB(url)
3     let content = await? read(conn)
4     return content
5 }
```

プログラム 3.3: async の削除

3.2.1 コード変換する関数の限定

以降でコード変換の対象となる await?を含むプログラムをプログラム 3.1 に示す．手順 1 では, await?の使用に適さない関数を変換対象から除外している．await?を for 文や while 文の中で使用した場合, 手順 3 で継続を切り出す際の CPS 変換において, 繰り返し文が再帰関数の呼び出しに変換されてしまい, かえって実行速度の低下を招くためである．したがって, ループを含む関数は await?を標準の await で置き換え, async 関数としたまま変換を終了する．プログラム 3.2 に示した関数は, 各地にあるセンサーと通信を行い, 温度のデータを取得して平均を求めるものである．4 行目の await?は for 文の中に含まれているため, この await?を標準の await で置き換えて変換を終了する．

await?を含むループの存在しない関数は変換を続行する．変換が全て終了すると関数内に await は残らないため, 手順 2 で async キーワードを取り除いておく．

3.2.2 await?式の選択と継続の切り出し

手順 3 では, 実行順で最初に登場する await?式を変換対象に選び, その await?式以降の継続を関数化する．切り出した継続は, 手順 6 でスローパスやファストパスを構築するために利用される．関数の形で切り出すことで, スローパスとファストパスの両方に登場する処理を共通化し, コードサイズの増加を抑えることができる．プログラム 3.1 に手順 3 までを施すとプログラム 3.4 が得られる．プログラム 3.4 では, プログラム 3.3 における 2 行目の await?式が変換対象に選ばれ, 以降の継続が関数 kont1() として切り出されている．

```
function sample(url) {
    const kont1 = (v1) => {
        let conn = v1
        let content = await? read(conn)
        return content
    }
    return kont1(await? connectDB(url))
}
```

プログラム 3.4: 継続の関数化

```
function if_sample(condition) {
    if (condition) {
        await? a()
    } else {
        await? b()
    }
    c()
}
```

プログラム 3.5: 条件分岐の節に await?が含まれる場合 (対応前)

```
function if_sample(condition) {
    if (condition) {
        await? a()
        c()
    } else {
        await? b()
        c()
    }
}
```

プログラム 3.6: 条件分岐の節に await?が含まれる場合 (対応後)

プログラム 3.5 に示した関数は, 条件分岐の節の中に await?を含み, 節を抜けた後に処理 c() が記述されている. このような関数を変換する際には, 手順 3 を進める前に特別な対応が必要となる. 継続を適切に切り出すため, 事前に節の外で行われる処理を条件分岐のそれぞれ節の末尾に追加した上で手順 3 を行う. プログラム 3.5 に事前の変換を施すとプログラム 3.6 が得られる.

```
1 function sample(url) {
2     const kont1 = (v1) => {
3         let conn = v1
4         let content = await? read(conn)
5         return content
6     }
7     const v1 = connectDB(url)
8     return kont1(await? v1)
9 }
```

プログラム 3.7: 代入文の挿入

```
1 function sample(url) {
2     const kont1 = (v1) => {
3         let conn = v1
4         let content = await? read(conn)
5         return content
6     }
7     const v1 = connectDB(url)
8     // v1 が Promise オブジェクトかどうかをチェック
9     if (v1 instanceof Promise) {
10        return kont1(await? v1)
11    } else {
12        return kont1(await? v1)
13    }
14 }
```

プログラム 3.8: パスを分岐させる if 文の挿入

3.2.3 代入文の挿入

手順 4 では, スローパスとファストパスを構築する前の準備として, 変換対象の await?が付与された式の結果を新たな変数に束縛する代入文を挿入する. スローパスまたはファストパスへ分岐する if 文において await?が付与された式の結果を利用するため, 一度変数に代入しておく必要がある. プログラム 3.1 に手順 4 までを施すとプログラム 3.7 が得られる. 7 行目において, 変換対象の await?が付与された式である connectDB(url) の結果が, 新たに定義した変数 v1 に束縛されている.

```
1  function sample(url) {
2      const kont1 = (v1) => {
3          let conn = v1
4          let content = await? read(v1)
5          return content
6      }
7      const v1 = connectDB(url)
8      if (v1 instanceof Promise) {
9          // スローパス
10         return v1.then(kont1)
11     } else {
12         // ファストパス
13         return kont1(v1)
14     }
15 }
```

プログラム 3.9: スローパスとファストパスの構築

3.2.4　パスを分岐させる if 文の挿入

手順 5 では, await?が付与された式が Promise オブジェクトを返すときにスローパス (then 節に対応) へ, それ以外の値を返すときにファストパス (else 節に対応) へ分岐する if 文を挿入する．instanceof 演算子を用いて変数の値を評価し, Promise オブジェクトであればスローパスへ, そうでなければファストパスへ分岐する．プログラム 3.1 に手順 5 までを施すとプログラム 3.8 が得られる．8 行目の if 文で, await?が付与された式の結果が代入された変数 v1 を instanceof 演算子で評価し, 適切なパスへの分岐を行う．

3.2.5　スローパスとファストパスの構築

手順 6 では, 手順 5 で挿入した if 文の then 節と else 節の中にそれぞれスローパスとファストパスを展開する．スローパスが実行されるとき, 変換対象の await?が付与された式は Promise オブジェクトを返したことになる．よって then 節の中では, その Promise オブジェクトの then() メソッドを呼び出し, 手順 3 で関数化した継続を渡す．ファストパスが実行されるとき, 変換対象の await?が付与された式は Promise オブジェクト以外の値を返したことになる．よって特別な対応は必要なく, else 節の中では手順 3 で関数化した継続に変換対象の await?が付与された式の結果を渡して実行する．プログラム 3.1 に手順 6 までを施すとプログラム 3.9 が得られる．手順 3 で切り出した継続 kont1() を, then 節の中 (10 行目) では Promise オブジェクト v1 の then() メソッドに渡している．一方 else 節の中 (13 行目) では, v1 に代入されている await?式が付与された式の結果を, 継続 kont1() に渡している．

```
1  function sample(url) {
2      const kont1 = (v1) => {
3          let conn = v1
4          const kont2 = (v2) => {
5              let content = v2
6              return content
7          }
8          const v2 = read(conn)
9          if (v2 instanceof Promise) {
10             // スローパス
11             return v2.then(kont2)
12         } else {
13             // ファストパス
14             return kont2(v2)
15         }
16     }
17     const v1 = connect(url)
18     if (v1 instanceof Promise) {
19         // スローパス
20         return v1.then(kont1)
21     } else {
22         // ファストパス
23         return kont1(v1)
24     }
25 }
```

プログラム 3.10: プログラム 3.1 の変換結果

3.2.6 再帰的な変換

手順 7 では, 手順 3 で関数化した継続に `await?`が含まれる場合, 再帰的に変換を施す. プログラム 3.9 中の継続 `kont1` には `await?`がまだ 1 つ含まれている (4 行目). 関数 `kont1()` を新たな変換対象として再帰的に変換を施すことで, 最終的にプログラム 3.10 が得られる. なお, プログラム 3.6 のように条件分岐の節に `await?`を含む関数の場合は, それぞれの節に対して再帰的に変換を施す必要がある.

```
function sample(url) {
    let _kont = (_v) => {
        let conn = _v
        let _kont2 = (_v2) => {
            let content = _v2
            return content
        }
        const _v2 = read(conn)
        if (_v2 instanceof Promise) {
            return _v2.then(_kont2)
        } else {
            return _kont2(_v2)
        }
    }
    const _v = connectDB(url)
    if (_v instanceof Promise) {
        return _v.then(_kont)
    } else {
        return _kont(_v)
    }
}
```

プログラム 3.11: コード変換器による変換

3.3　コード変換器の実装

3.2 節で述べた手法に従い, await?を含むプログラムを純粋な JavaScript コードへ変換する変換器を開発した. 変換器は JavaScript トランスパイラの Babel[*1]を利用して開発した. Babel は主に ECMAScript2015 以降の JavaScript コードを, 古いブラウザや環境で実行できる後方互換性のあるコードに変換する用いられるツール群である.

Babel を利用して抽象構文木を操作するためには, 入力として純粋な JavaScript コードを与える必要がある. 先に述べたコード変換の手法において与えられる入力は await?を含むプログラムであり, 純粋な JavaScript コードではない. そのため, Babel に与える入力は await を含む純粋な JavaScript コードとし, await の部分に await?が書かれているものとみなして変換を行う方法を採用した. プログラム 3.1 に含まれる await?を await に書き換えた上でコード変換器に入力するとプログラム 3.11 が得られる. 新たに定義する変数の名前に関しては, 既に定義されている変数の名前と重複しないように決定しているためにプログラム 3.10 の例とはわずかに異なるが, 制御構造については同じものが得られる.

[*1] https://babeljs.io/

第 4 章

実験

本章では, 3 章で提案した await?の性能を評価する実験について述べる．本実験では, await?の性能について以下の 3 つの評価指標を設け, それぞれに対応する実験を行った.

1. Promise オブジェクト以外を返す式に付与した場合の実行時間
2. Promise オブジェクトを返す式に付与した場合の実行時間
3. コード変換によって得られた JavaScript コードのサイズ

await?は標準の await と比較して，Promise オブジェクト以外を返す式に付与されていた場合はより高速に動作し，Promise オブジェクトを返す式に付与されていた場合にも大きなオーバヘッドは生じないことが期待される．1 つ目の評価指標はファストパスを通過することによる速度改善を, 2 つ目の評価指標はスローパスを通過することによるオーバヘッドを確認するためのものである．JavaScript コードを送受信するようなシーンを考慮した場合, コードサイズがあまりに大きいものはシステムの速度を低下させるおそれがある．3.2 節で提案したコード変換では, 継続を関数の形で切り出すことによりコードサイズの増加を抑制している．そこで, 3 つ目の評価指標として, コード変換を行うことによるコードサイズの変化についても着目する.

本章では, 初めに実験のために新たに準備したベンチマークの作成方法について述べる．そして, コード変換によるコードサイズの変化を比較するために用意した, 3.2 節で提案したコード変換のバリエーションについて述べる．最後に実験結果について考察を行う.

4.1 ベンチマーク

使用する await?の割合が変化することによる実行時間の変化を調べるため, 含まれる await または await?の割合が異なる複数のベンチマークを準備した．同期関数の呼び出しに await を付与しているベンチマークは我々が知る限り存在しないため, まずはそのようなベンチマークを開発した．Marr らの Cross-Language Compiler Benchmarking [1] に含まれるベンチマークを元に, 一部の関数宣言文に async キーワードを付与し, その内部の関数呼び出し式に await を付与することにより作成した．そして, これらのベンチマークに含まれる await

```
class Towers extends Benchmark {
    constructor() {/* 省略 */}

    benchmark() {
        this.piles = new Array(3)
        this.buildTowerAt(0, 13)
        this.movesDone = 0
        this.moveDisks(13, 0, 1)
        return this.movesDone
    }

    // 省略
}
```

プログラム 4.1: マイクロベンチマーク Towers(一部抜粋)

```
class Towers extends Benchmark {
    constructor() {/* 省略 */}

    async benchmark() {
        this.piles = new Array(3)
        await wrap(this.buildTowerAt(0, 13))
        this.movesDone = 0
        await wrap(this.moveDisks(13, 0, 1))
        return this.movesDone
    }

    // 省略
}

function wrap(x) {
    // true のとき，Promise で包んで返す
    if (true) {
        return new Promise((resolve) => resolve(x))
    } else {
        return x
    }
}
```

プログラム 4.2: マイクロベンチマーク Towers を元に作成したベンチマーク (一部抜粋)

を await?が書かれているものとみなし, コード変換器を用いて変換したベンチマークも作成した. また各ベンチマークには, 各関数が Promise オブジェクトを返すかどうかを制御できるよう, 関数呼び出しをラップする関数 wrap() を新たに定義した. 関数 wrap() は, 受け取った引数を内部のフラグの値が true のとき Promise オブジェクトで包んで返却し, false のときそのまま返却する. 作成したベンチマーク 1 つにつき, await と await?のどちらを利用するか, 各関数が Promise オブジェクトを返すかそれ以外を返すかを選択できるため, 合計 4 種類のバリエーションが存在することになる.

Marr らの Cross-Language Compiler Benchmarking には, 9 個のマイクロベンチマーク (Bounce, List, Mandelbrot, Nbody, Permute, Queeens, Sieve, Storage, Towers) と 5 個のマクロベンチマーク (CD, DeltaBlue, Havlak, Json, Richards) が含まれる. ベンチマークの例として, プログラム 4.1 に示したマイクロベンチマーク Towers に対して, 上記の変更を加えて作成したものをプログラム 4.2 に示す. 関数 benchmark() の宣言文に async を付与し, 内部の関数呼び出し式に await を付与している. 関数 wrap() 内部のフラグの値は true であるため, buildTowerAt() メソッドと moveDisks() メソッドは Promise オブジェクトを返す関数とみなすことができる. 仮に buildTowerAt() メソッドや moveDisks() メソッドが非同期関数であった場合でも, 同期関数であった場合と同じように動作する.

async を付与し, 内部の関数呼び出しに await を付与する関数の数を変えることで, 元となるベンチマークから複数のベンチマークを作成した. ここで, 正しく動作するベンチマークを作成するには, 上記のように非同期関数の呼び出しにも対応するための変更を加える関数を適切に選択する必要がある. 関数宣言文に async を付与すると, その関数は Promise オブジェクトを返すようになるため, 変更を加えた関数を呼び出す関数は, 同様に非同期関数の呼び出しにも対応している必要がある. そうでない場合, 元のベンチマークは Promise オブジェクトが返却されることを想定して作成されていないため, Promise オブジェクトが適切に処理されず, 作成したベンチマークは正しく動作しない. 以上のことから, ベンチマーク内の関数の呼び出し関係を考慮し, 呼び出す側の関数が非同期関数の呼び出しにも対応している状態を保ちながら, 関数に変更を加えていけば良いことがわかる.

この方針に従って複数のベンチマークを簡便に作成するために, 各関数にレベルと名付けた整数値を割り当てて管理し, レベル毎にまとめて非同期関数の呼び出しに対応させる方法を採用した. レベルは次の規則に従って割り当てた.

1. 内部に関数呼び出しの無い関数のレベルは 0 とする.
2. 内部に関数呼び出しのある関数のレベルは, 呼び出している関数のレベルの最大値 +1 とする.
3. 外部ライブラリの関数のレベルは 0 とする.
4. ある関数群に再帰構造が存在する場合, 関数群に含まれる各関数のレベルは, 関数群全体で呼び出している再帰に関わらない関数のレベルの最大値 +1 とする.
5. 内部でコールバック関数を呼び出している関数のレベルは, 受け取る可能性があるコールバック関数と, それ以外に呼び出している関数のレベルの最大値 +1 とする.

表 4.1: 作成したベンチマークの詳細

元の ベンチマーク	レベル	関数の個数			呼び出し回数		
		同期	非同期	合計	同期	非同期	合計
Bounce	1	1	2	3	2139000	7501500	9640500
	2	2	1		9639000	1500	
List	1	1	4	5	4213500	4281000	8494500
	2	4	1		8493000	1500	
Storage	1	1	2	3	4096000	5462000	9558000
	2	2	1		9557000	1000	
Towers	1	2	4	6	9837600	9830400	19668000
	2	4	2		14752800	4915200	
	3	5	1		19667400	600	
DeltaBlue	1	40	39	79	3516523	3012710	6529233
	2	50	29		5640776	888457	
	3	56	23		5964961	564272	
	4	58	21		6169128	360105	
	5	61	18		6205140	324093	
	6	63	16		6241152	288081	
	7	64	15		6241157	288076	
	8	68	11		6421204	108029	
	9	76	3		6529227	6	
	10	78	1		6529232	1	
Richards	1	18	21	39	30636200	23225700	53861900
	2	31	8		38379100	15482800	
	3	33	6		41703700	12158200	
	4	36	3		47282700	6579200	
	5	37	2		53861700	200	
	6	38	1		53861800	100	
Json	1	33	33	66	4698000	6704700	11402700
	2	43	23		7885700	3517000	
	3	47	19		10020600	1382100	
	4	53	13		10604000	798700	
	5	61	5		10992100	410600	
	6	62	4		10992400	410300	
	7	65	1		11402600	100	

このようにレベルを割り当てることで, すべての関数は自分のレベルよりも高い関数を呼び出すことはない. そのため, あるレベル以上の関数すべてに対し, `async` を付与して内部の関数呼び出しに `await` を付与するという変更を加えることで, 正しく動作するベンチマークが作成できる. 以降では, 元のベンチマークのレベル n 以上の関数すべてに対して非同期関数の呼び出しに対応する変更を加えて作成したベンチマークを”レベル n のベンチマーク”と呼ぶ. このとき, レベル 1 のベンチマークは, ベンチマーク中に存在するすべての関数呼び出しに `await` が付与されていることになる. また, Towers を元に作成したレベル 1 のベンチマークという意味で Towers1 という表現を用いる. 他のベンチマークやレベルに対しても同様である.

コンストラクタの中に関数呼び出しが存在する場合は特別な修正を行っている. コンストラクタの宣言文に `async` を付与することはできないため, 関数呼び出しのある部分を記述したメソッド `init()` を新たに定義し, コンストラクタの実行直後に実行されるように変更を加えている. 新たに定義した関数 `init` には `async` を付与することができ, 1 以上のレベルを割り当てることができる. また, 外部ライブラリの関数を呼び出すにあたってコールバック関数を引数として渡す必要がある場合は, そのコールバック関数についてのみ変更を加えなかった. コールバック関数の内部で変更を加えた関数を呼び出している場合は, 変更を加える前の関数を別名で定義し利用することで解決した.

以上の手順を踏んで作成したベンチマークについて, `async` を付与した関数とそうでない関数の個数, 実行時の関数呼び出しの回数を表 4.1 に示す. Marr らの Cross-Language Compiler Benchmarking に含まれるマイクロベンチマークのうち, プログラム中の全ての関数において関数呼び出しを含む `for` 文や `while` 文が存在する Mandelbrot, Nbody, Permute, Queeen, Sieve は実験の対象から除外した. このようなベンチマークを元に作成したベンチマークは, `await` を `await?`とみなしてコード変換を行った場合, `await?` がコード変換によって標準の `await` に置き換えられ, 元のプログラムと同一になってしまうためである. また, マクロベンチマーク CD 及び Havlak については, これらを元に作成したベンチマークが実行時エラーにより動作しなかったため, 実験の対象から除外している.

以降では, `await` を付与したベンチマークに対し, `await?`が書かれているとみなしてコード変換することで得られるベンチマークを”`await?`を付与した”ベンチマークと表現する.

4.2 比較に用いたコード変換

本研究で提案した手法は, 継続を関数化して一部の処理を共通化することによりコードサイズの増加を抑制している. 本節では, コード変換の方法を変えることによるコードサイズへの影響を評価するために用意した, 3.2 節で述べた手法のバリエーションについて説明する. まず, 継続を切り出す際に関数化を行わず, スローパス及びファストパスにそのまま展開するコード変換について述べる. この変換は, 継続を関数化する工夫を省いたことで, 新たに関数を生成しない代わりに, コードサイズがより大きく増加する. 次に, 継続を関数化する変換と関数化しない変換を混合させた変換について述べる. この変換は, 混合割合を表すハイパーパラメータを調整することにより, コードサイズの増加具合を段階的に変化させることができる.

```
1  function sample(url) {
2      const v1 = connectDB(url)
3      if (v1 instanceof Promise) {
4          // スローパス
5          return v1.then((v1) => {
6              let conn = v1
7              let content = await? read(v1)
8              return content
9          })
10     } else {
11         // ファストパス
12         let conn = v1
13         let content = await? read(v1)
14         return content
15     }
16 }
```

プログラム 4.3: スローパスとファストパスの構築 (関数化なし)

4.2.1　継続の関数化を行わない変換

3.2 節で述べたコード変換の手順を一部変更し, 継続を切り出す際に関数化を行わない変換について述べる．この変換は, 3.2 節で述べたコード変換の手順 6 にもう 1 つ操作を追加することで達成される．手順 6 を終えた時点で, 変換中のプログラムにはスローパスとファストパスが構築されている．ここで, それぞれのパスの中に登場する継続を表現した関数を利用している部分に, 関数の処理の中身をそのまま展開する．3.2 節で用いていたプログラム 3.1 に対して以上の操作を施すと, プログラム 4.3 が得られる．スローパスとファストパスに継続を表す関数 `kont1()` の中身をそのまま展開したことで, 新たな関数宣言がなされることはない．その代わり, スローパスとファストパスの両方に `await?`が残るため, それぞれに対して再帰的に変換を施す必要がある．`then` 節と `else` 節のそれぞれを新たな変換対象の関数とみなした上で, 再帰的な変換を施すことで最終的にプログラム 4.4 が得られる．`then` 節 (4-19 行目) と `else` 節 (21-34 行目) のそれぞれに, 2 つ目の `await?`に対応するスローパスとファストパスが構築されている．

変換の手順からわかるように, 変換対象となる関数において実行時に通過する `await?`の最大個数に対してコードサイズが 2 の冪乗のオーダーで増加する．3.2 節で述べたコード変換は, スローパスとファストパスに共通する処理を関数化できていたので, 関数に含まれる `await?`の個数の増加に伴うコードサイズの増加がより小さい．

```
function sample(url) {
    const v1 = connectDB(url)
    if (v1 instanceof Promise) {
        // スローパス 1
        return v1.then((v1) => {
            let conn = v1
            let v2 = read(v1)
            if (v2 instanceof Promise){
                // スローパス 2
                return v2.then((v2) => {
                    let content = v2
                    return content
                })
            } else {
                // ファストパス 2
                let content = v2
                return content
            }
        })
    } else {
        // ファストパス 1
        let conn = v1
        let v2 = read(v1)
        if (v2 instanceof Promise){
            // スローパス 2
            return v2.then((v2) => {
                let content = v2
                return content
            })
        } else {
            // ファストパス 2
            let content = v2
            return content
        }
    }
}
```

プログラム 4.4: プログラム 3.1 の変換結果 (関数化なし)

```
async function sample2(url) {
    let conn = await? connectDB(url)
    let content = await? read(conn) // 最初の変換対象となる await?
    let result = await? parseResult(content)
    return result
}
```

プログラム 4.5: コード変換の対象となるプログラム例 2

4.2.2　継続の関数化を部分的に行う変換

3.2 節で述べた継続を関数化する変換と 4.2.1 節で述べた継続を関数化しない変換を混合した変換について述べる．この変換では, 基本的には継続を関数化しない変換を行うが, 変換対象の関数がハイパーパラメータ N を超える await?を含む場合に, 実行順でちょうど中央に位置する await?式を対象に継続を関数化する変換を行う．このような手順を踏むことで, await?のうち半分は継続を表現する関数に含まれ, 残りはもともと変換対象だった関数の直下に残る．この手順を最初に行うことで, 継続を関数化しない変換の対象となる関数に含まれる await?の数は常に N 以下となり, N を変化させることでコードサイズの増加具合を調節することができる．

コード変換の手順は以下の通りである．ここで, 変換対象の関数に含まれる await?の数を n とする．

1. ループの中に await?が含まれる場合，関数内のすべての await?を標準の await で置き換え，変換を終了する．
2. 関数宣言文から async キーワードを取り除く．
3. (a) $n \leq N$ のとき: 実行順で最初に登場する await?式を変換の対象に選び，選択した await?式以降の継続を切り出す．
   (b) $n > N$ のとき: 実行順で $\lceil n/2 \rceil$ 番目に登場する await?式を変換の対象に選び，選択した await?式以降の継続を切り出し関数化する．
4. await?が付与された式の結果を変数に束縛する代入文を挿入する．
5. await?の対象である式の結果が Promise オブジェクトかどうかによって分岐する if 文を挿入する．
6. スローパスとファストパスを構築する．
7. (a) $n \leq N$ のとき: then 節 (スローパス) と else 節 (ファストパス) が await?を含むなら, 節の内部を新たな変換対象の関数とみなして再帰的に処理する．
   (b) $n > N$ のとき: 継続を表現した関数を新たな変換対象として再帰的に処理する．その後, 変換対象とした await?式よりも前の部分を新たな変換対象の関数とみなして再帰的に処理する．

```
1  function sample2(url) {
2      let conn = await? connectDB(url)
3      const kont1 = (v2) => {
4          let content = v2
5          let result = await? parseResult(content)
6          return result
7      }
8      const v2 = read(conn)
9      if (v2 instanceof Promise) {
10         // スローパス 2
11         return v2.then(kont1)
12     } else {
13         // ファストパス 2
14         return kont1(v2)
15     }
16 }
```

プログラム 4.6: プログラム 4.5 の変換の途中経過 (ハイパーパラメータ N は 1)

ハイパーパラメータ N を 1 としたとき, プログラム 4.5 を変換するとプログラム 4.7 が得られる. 関数 `sample2()` に含まれる `await?`の個数は 3 ($> N$) であるから, まずは継続を関数化する変換を行う. 変換対象となる `await?`式は実行順で $\lceil 3/2 \rceil = 2$ 番目のもの (3 行目) である. 2 番目の `await?`式に対する変換を終えると, プログラム 4.6 が得られる. 残る 2 つの `await?`(2 行目及び 5 行目) についても再帰的に変換する. まず, 変換した 2 番目の `await?`式以降の継続を表現した関数 `kont1()` を新たな対象として変換を行う. この関数に含まれる `await?`の個数は 1 ($\leq N$) であるから, 継続を関数化しない変換を行う. 変換対象となる `await?`式は実行順で最初のもの (プログラム 4.6 では 5 行目) である. 次に, 変換した 2 番目の `await?`式よりも前の部分を新たな対象として変換を行う. この部分に含まれる `await?`の個数は 1 ($\leq N$) であるから, 継続を関数化しない変換を行う. 変換対象となる `await?`式は実行順で最初のもの (プログラム 4.6 では 2 行目) である. それぞれの変換を終えると, プログラム 4.7 が得られる.

この変換では, N が大きいほど継続を関数化しない変換が選択される割合が増え, コードサイズが増加する. 極端な場合として, $N = \infty$ の場合には継続を関数化しない変換と一致する. より正確には, N が変換対象の関数に含まれる `await?`の個数と同じあるいはそれより大きいとき, 継続を関数化しない変換と一致する. また, $N = 0$ の場合には継続を関数化する変換と一致する.

```
function sample2(url) {
    const v1 = connectDB(url)
    if (v1 instanceof Promise) {
        // スローパス 1
        return v1.then((v1) => {
            let conn = v1
            const kont1 = (v2) => {
                let content = v2
                const v3 = parseResult(content)
                if (v3 instanceof Promise) {
                    // スローパス 3
                    return v3.then((v3) => {
                        let result = v3
                        return result
                    })
                } else {
                    // ファストパス 3
                    let result = v3
                    return result
                }
            }
            const v2 = read(conn)
            if (v2 instanceof Promise) {
                // スローパス 2
                return v2.then(kont1)
            } else {
                // ファストパス 2
                return kont1(v2)
            }
        })
    } else {
        let conn = v1
        const kont1 = (v2) => {
            let content = v2
            const v3 = parseResult(content)
            if (v3 instanceof Promise) {
                return v3.then((v3) => {
                    let result = v3
                    return result
                })
            } else {
                let result = v3
                return result
            }
        }
        const v2 = read(conn)
        if (v2 instanceof Promise) {
            return v2.then(kont1)
        } else {
            return kont1(v2)
        }
    }
}
```

プログラム 4.7: プログラム 4.5 の変換結果 (ハイパーパラメータ N は 1)

表 4.2: 実行したベンチマークのパラメータ設定

ベンチマーク	extraArgs
Bounce	1500
List	1500
Storage	1000
Towers	600
DeltaBlue	12000
Richards	100
Json	100

4.3 実験結果

作成したベンチマークを用い, 本章の冒頭で述べた 3 つの評価指標に対応する実験を行った. 実行時間の測定はベンチマーク 1 つにつき 60 回行い, 最初の 10 回を除いた結果の平均値を求めた.

各ベンチマークを実行するにあたって設定したパラメータを表 4.2 に示した. このパラメータは実行するベンチマークで扱う問題の大きさを決定するものであり, 値が大きいほど実行時間も増加する. ベンチマークの大半は問題の大きさが変更できず, 一度の計測で固定のベンチマークを何度実行するかというイテレーションの回数を表している. 本実験に利用したベンチマークのうち DeltaBlue についてのみ, 問題そのものの大きさを調整するパラメータとして機能している. パラメータの値を決定するにあたっては, Marr らの Cross-Language Compiler Benchmarking の GitHub リポジトリ[*1]を参照した.

実験には, クロック周波数 3.2GHz, コア数 8 の CPU を持つ Apple M1 Pro と 16GB の LPDDR5 を搭載した PC を用い, プログラムはすべて Node.js 上で実行した. Node.js のバージョンは v18.16.1 であった.

`Promise` オブジェクト以外を返す式に付与した場合の実行時間

作成したベンチマーク 1 つにつき, `await` と `await?`のどちらを付与するか, 各関数が `Promise` オブジェクトを返すかそれ以外を返すかを選択できるため, 合計 4 種類のバリエーションが存在する. 作成した全 32 個のベンチマーク, 4 種類のバリエーションのそれぞれについて実行時間を計測した. `await?`を利用する場合は, `await` を含むベンチマークを, `await` と `await?`とみなした上で 3.2 節で述べた手法で変換したのち実行した.

`Promise` オブジェクト以外を返す式に `await` または `await?`を付与した場合の実行時間のグラフを図 4.1 に示した. グラフの縦軸には, 元となった Marr らの Cross-Language Compiler

[*1] `https://github.com/smarr/are-we-fast-yet/blob/master/rebench.conf`

Benchmarking に含まれるベンチマークに対する実行時間の割合をとった．横軸には, ベンチマークを実行した際の関数呼び出しの回数のうち, 非同期関数を呼び出したものの割合をとっている．グラフ上の各点が各レベルのベンチマークに対応しており, より右側の点ほどレベルが低いベンチマーク, すなわちより多くの `await` または `await?`を含み, 非同期関数を呼び出した割合が大きいベンチマークの実験結果を表している．2 つ目の実験と合わせて, 各ベンチマーク, 4 種類のバリエーションの実行時間は表 4.3 にまとめた.

`Promise` オブジェクト以外を返す式に付与した場合, `await?` は標準の `await` と比較してベンチマークの実行時間が概ね短縮された．短縮されたものの中には, List1 や Towers1 のように, 10% 近くまで短縮されるものもあった．これらの元になったベンチマークはマイクロベンチマークであり, プログラムの中核に再帰構造を含むものであった．一方で実行時間が数 % 増加したベンチマークも存在した．レベルの高いベンチマークに関しては, 全体に占める非同期関数の呼び出し回数が少なく, `await?`と標準の `await` の差が結果に現れなかったものと考えられる．また, Bounce と Storage に関してはレベルに依らず速度改善が見られなかった．こちらのベンチマークに関しては, 関数の数が少ない上に `await` または `await?`を含むループのある関数の割合が高く, 変換によって標準の `await` に置き換えられる `await?`の数が大半を占め, 互いの共通部分が多くなったことが原因と考えられる．ベンチマークのレベルによる違いを見ると, 先に述べた Bounce と Storage を除いて, 概ねレベルが低くなるほどより実行時間が短縮されていることがわかる．レベルが低くなるほど付与されている `await?`ないし `await` の数が増加するため, 両者の速度差がより顕著に現れているものと考えられる.

元となったベンチマークと比較すると, `await?`を付与することによって最大で 2290% の実行時間増加がみられた．レベルが低く `await?`の数が多いベンチマークに関しては 1000% を超える実行時間の増加がみられており, このようなベンチマークに関しては `await` との差を評価するためには有効であっても, 元のベンチマークとの比較では意味を持たないと考えられる．反対にレベルの高いベンチマークに関しては 1% 強の実行時間改善がみられるものもあったが, こちらに関しては付与された `await?`の数が少なく誤差の範囲であるといえる．マクロベンチマーク DeltaBlue, Richards, Json について, 非同期関数の呼び出し回数の割合が 10% 程度の部分 (DeltaBlue2-3, Richards4, Json3-4) に注目すると, およそ 300% から 1000% の実行時間増加がみられる．標準の `await` と比較すると, 13% 程度の速度改善となる．3 つの中では Json が最も実行時間増加を抑えており, これはベンチマーク中に存在する `await?`を含むループのある関数の数が少ないためであると考えられる．極端な結果として, 再帰構造を持つベンチマークである List と Towers については, 付与する `await?`の数が増加しても最大で 234% の実行時間増加に留まった．これは, 再帰構造がコード変換中の CPS 変換に自然に組み込めることが要因と考えられる．反対に標準の `await` を再帰構造を持つ関数で利用した場合は大きな速度低下がみられ, 結果として List と Towers では `await` と `await?`の間に大きな差がみられた.

Promise オブジェクトを返す式に付与した場合の実行時間

Promise オブジェクトを返す式に await または await?を付与した場合の実行時間のグラフを図 4.2 に示した．Promise オブジェクトを返す式に付与した場合, await?は標準の await と概ね同程度の実行時間を達成した．最も大きく実行時間が増加したもので 108.8%, 最も短縮されたものでで 93.15% という結果であり, Promise オブジェクトを返す式に付与した場合, 両者に大きな差はないといえる．これは，await?を利用した場合でも await を利用した場合でも Promise オブジェクトを処理しているためであると考えられる．

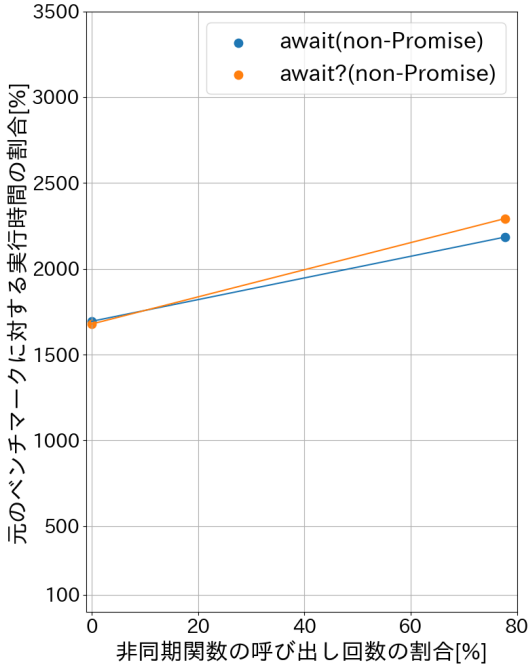
コード変換によって得られた JavaScript コードのサイズ

await?を付与し, Promise 以外を返すようにしたベンチマーク List1, Towers1, Richards1, Json1 を対象に, 4.2.2 節で述べたハイパーパラメータ N を導入したコード変換の手法を用いて変換を行い, N の変化に伴う実行時間とコードサイズの変化を計測した．ハイパーパラメータ N が意味を持つのは, ベンチマーク中に await?を含むループを持たず, かつ await?を複数個含む関数がある程度存在する場合である．そのため, この条件を満たす上記の 4 つのベンチマークを評価対象とした．ハイパーパラメータ N が 1 つの関数が含む await?の最大個数を上回った場合, それ以上 N の値を増やしても変換によって得られるベンチマークは変わらない．各ベンチマークにおいて, 変換結果が変化しなくなるまで N を 1 ずつ増やしていきながら変換を施し, 得られたベンチマークのそれぞれについてコードサイズと実行時間を計測した．

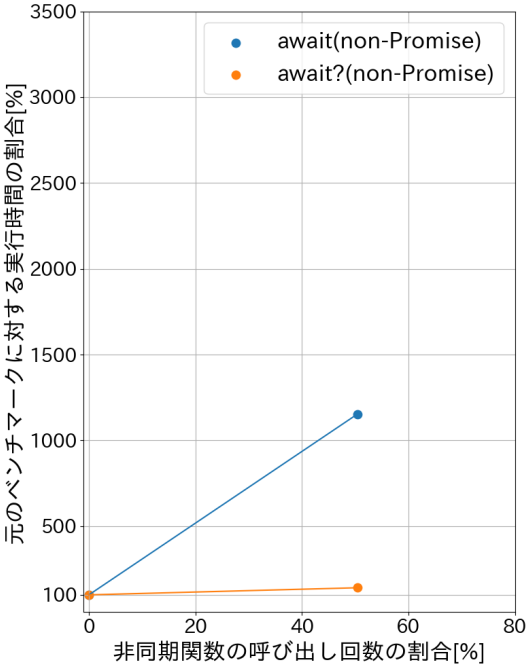
N の値の変化に伴う実行時間およびコードサイズの変化を図 4.3 に示した．$N = 0$ の場合が本研究で提案したコード変換, すなわち継続の関数化を常に行うコード変換に対応している．なお, ベンチマーク Richards1 には await?を 20 個含む関数が含まれており, $N = 4$ 以降の変換でコードサイズが爆発し実行困難となったため, $N = 3$ までの結果を掲載している．コードサイズに関しては, どのベンチマークも N が大きくなればなるほど増加しており, List1, Towers1, Richards1 については, $N = 0$ の場合と N 最大の場合を比較すると 2 倍から 3 倍程度の増加がみられる．また Richards1 については,await?を 20 個含む関数の存在により, $N = 3$ の時点で $N = 0$ の場合の 1768 倍にまでコードサイズが増加している．一方, 実行時間に関しては N の値が大きいほど短縮される傾向にあった．List1 と Towers1 に関しては, $N = 0$ の場合と比較して, 最も大きい N の場合, すなわち継続の関数化を全く行わない変換を施した場合に実行時間がそれぞれ 83.7%, 65.0% に短縮されている．関数の変換に際して関数化を行わない場合, 継続を表現する関数の呼び出しが存在しない．List1 や Towers1 は特定の 1 つの関数を何度も呼び出す構造となっており, その関数の変換に際して関数化を行わなかったことにより, 関数呼び出しによるオーバヘッドが大きく削減されたものと考えられる．また, Richards1 と Json1 については数 % の改善に留まった．このことから, 継続の関数化を行うことによって, 実行時間の増加を最大で 1.5 倍程度に抑えつつ, コードサイズを半分以下に削減することができることがわかった．

結果のまとめ

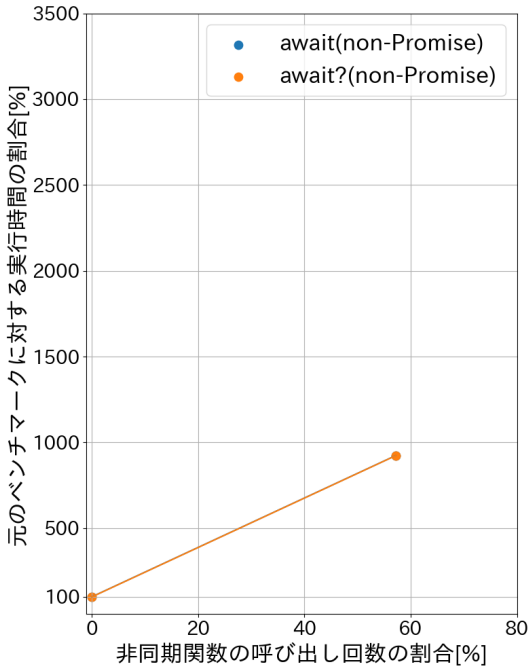
本研究で提案した `await?` を `Promise` オブジェクト以外を返す式に付与した場合, 標準の `await` と比較してベンチマークの実行時間が短縮された. `Promise` オブジェクトを返す式に付与した場合, 標準の `await` と比較してベンチマークの実行時間は概ね同程度であった. 元となったベンチマークと比較すると, `Promise` オブジェクト以外を返す式に `await?`を付与することにより最大で 2290% の実行時間増加がみられた. 実用性を考え, マクロベンチマーク DeltaBlue, Richards, Json について非同期関数の呼び出し回数の割合が 10% 程度の部分に着目すると, 実行時間の増加はおよそ 300% から 1000% であった. これを標準の `await` と比較した場合は, 13% 程度の速度改善となる. また, 継続の関数化を行うことによって, 実行時間の増加を最大で 1.5 倍程度のオーバヘッドが生じるが, コードサイズを半分以下に削減できることがわかった.



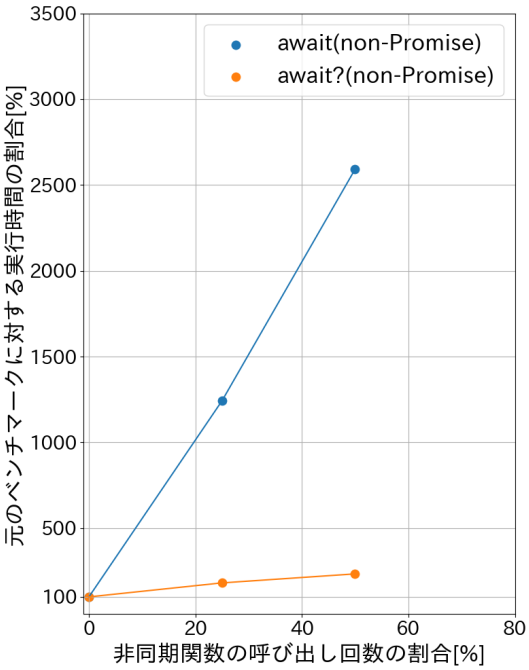
(a) Bounce



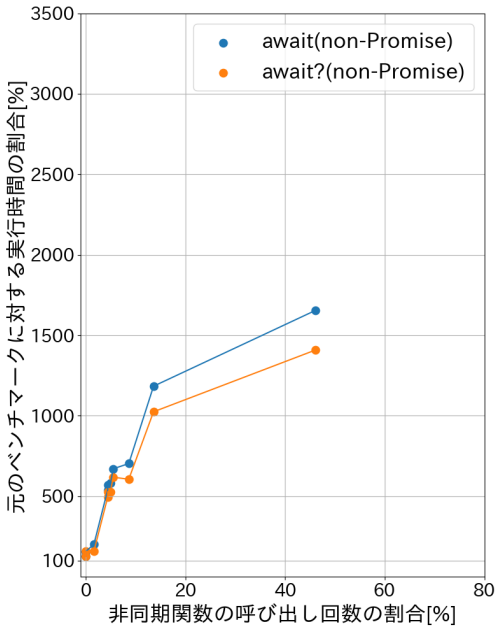
(b) List



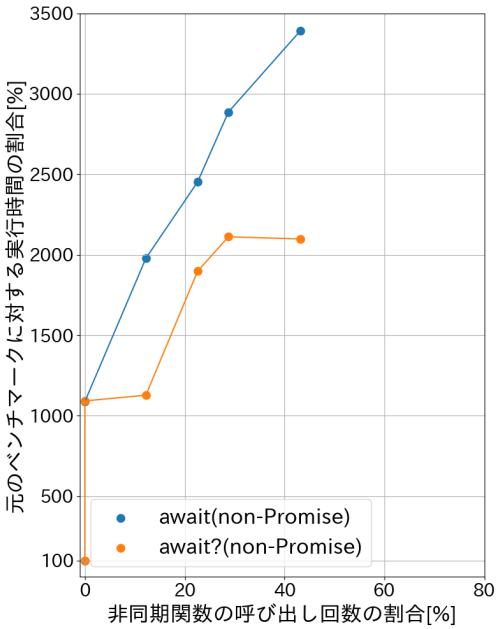
(c) Storage



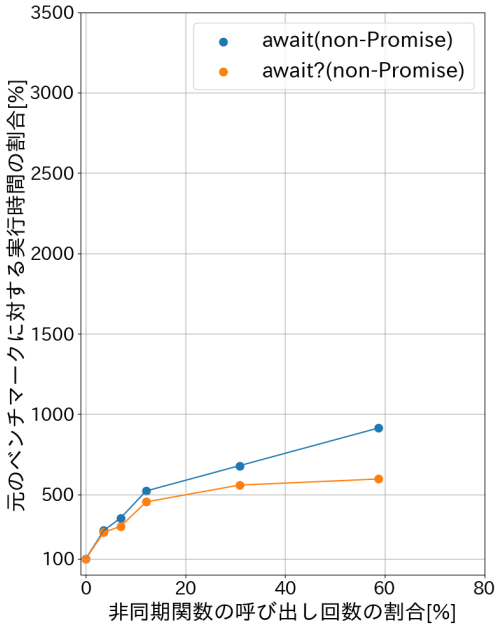
(d) Towers



(e) DeltaBlue

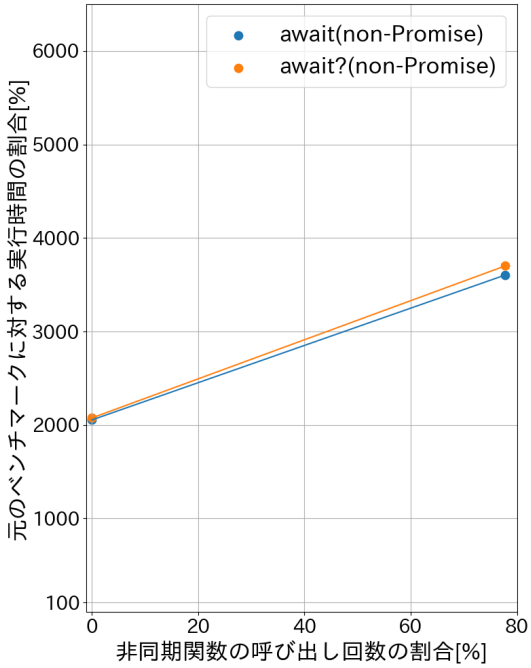


(f) Richards

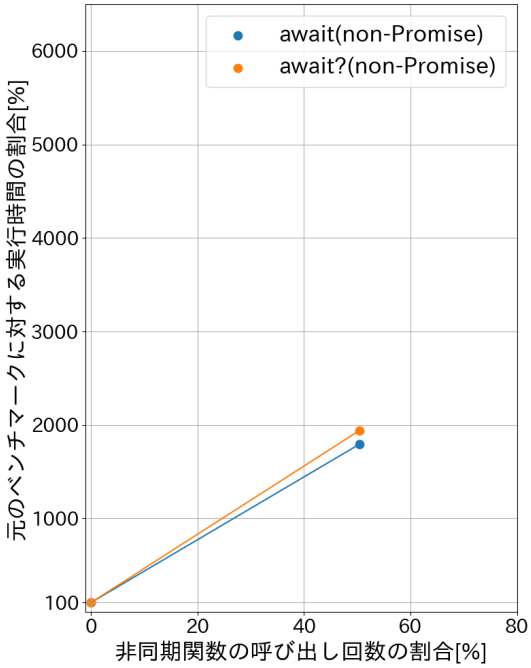


(g) Json

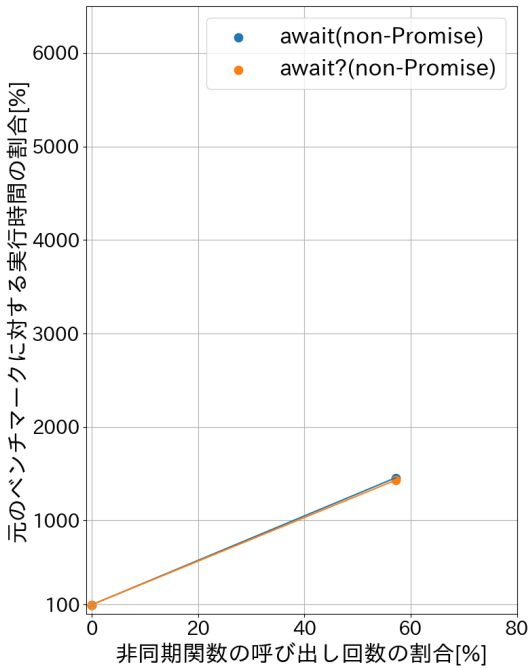
図 4.1: Promise 以外を返す式に `await` または `await?`を付与した場合の実行時間



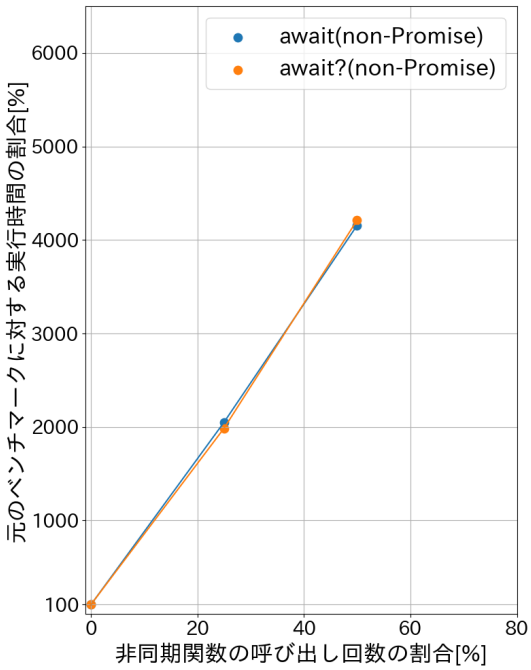
(a) Bounce



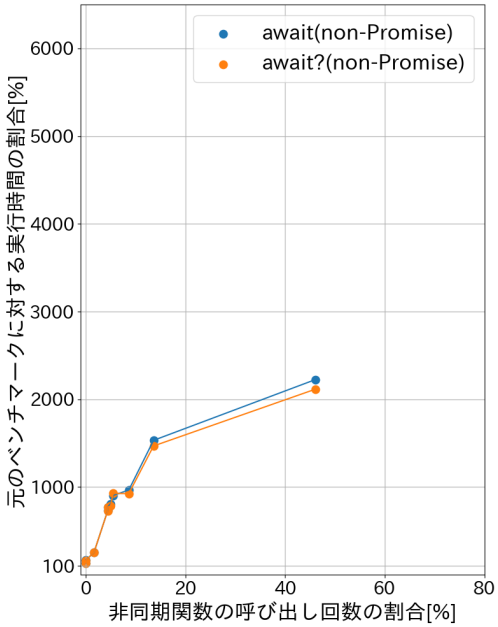
(b) List



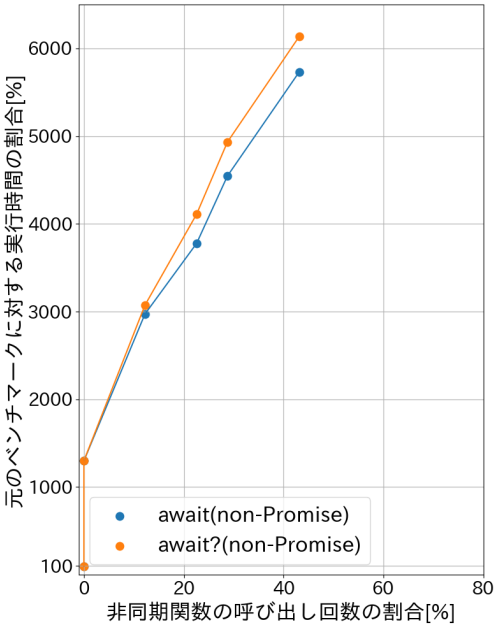
(c) Storage



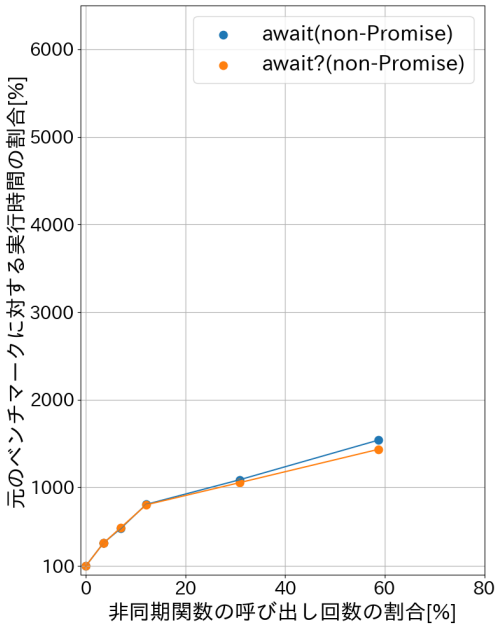
(d) Towers



(e) DeltaBlue



(f) Richards

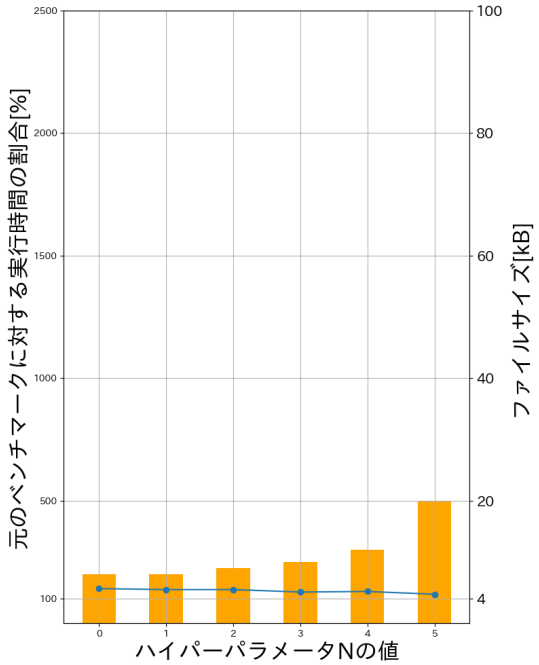


(g) Json

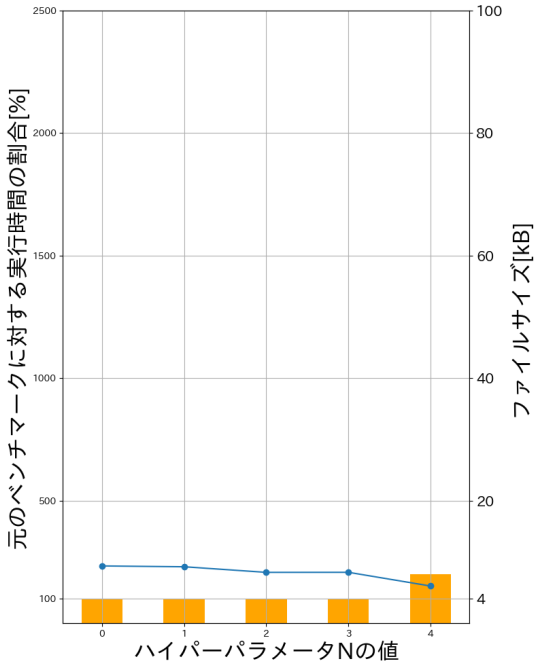
図 4.2: Promise を返す式に `await` または `await?`を付与した場合の実行時間

表 4.3: 作成したベンチマークの実行時間 [ms]

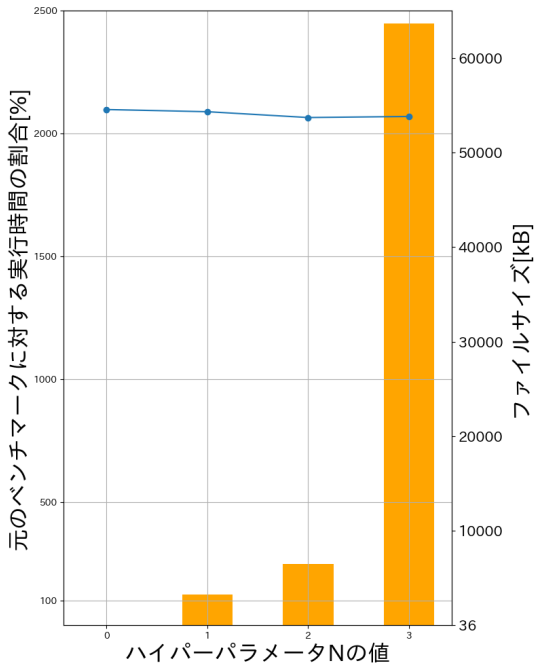
元のベンチマーク	レベル	non-Promise			Promise		
		await	await?	await?/await	await	await?	await?/await
Bounce	1	546.7	573.7	**104.9%**	901.8	926.1	102.7%
	2	423.9	419.9	99.06%	513.8	519.2	101.0%
List	1	510.4	62.6	12.3%	794.0	858.8	108.2%
	2	44.4	44.3	99.8%	44.8	45.0	101%
Storage	1	713.2	712.7	99.93%	1125	1106	98.25%
	2	77.5	76.9	99.2%	76.4	76.8	101%
Towers	1	983.1	88.7	**9.02%**	1576	1597	101.3%
	2	471.2	68.8	14.6%	777.8	753.4	96.86%
	3	37.9	37.9	100%	37.8	38.2	101%
DeltaBlue	1	318.1	270.8	85.12%	427.9	406.9	95.08%
	2	227.5	197.0	86.57%	294.6	282.3	95.82%
	3	135.2	116.5	86.17%	185.7	177.9	95.78%
	4	128.9	118.9	92.21%	174.0	178.5	102.6%
	5	111.9	101.7	90.89%	155.7	150.4	96.63%
	6	109.5	101.56	92.72%	148.6	148.6	100.0%
	7	103.8	95.0	91.6%	140.7	140.1	99.55%
	8	39.1	30.9	78.8%	48.7	49.2	101%
	9	30.0	30.2	101%	32.0	31.4	98.3%
	10	24.8	24.4	98.3%	25.5	25.7	101%
Richards	1	3183	1970	61.89%	5382	5761	107.0%
	2	2711	1983	73.15%	4269	4632	108.5%
	3	2303	1782	77.39%	3547	3860	**108.8%**
	4	1858	1059	57.01%	2790	2885	103.4%
	5	1022	1026	100.4%	1220	1221	100.0%
	6	92.7	94.4	102%	92.9	93.1	100%
Json	1	636.1	415.1	65.25%	1071	998.1	**93.15%**
	2	472.7	388.9	82.27%	755.4	732.4	96.96%
	3	363.3	315.8	86.92%	559.9	556.5	99.39%
	4	246.2	209.9	85.25%	367.8	374.6	101.8%
	5	192.0	184.4	96.07%	254.6	253.5	99.58%
	6	190.7	184.1	96.52%	255.2	255.7	100.2%
	7	69.6	69.0	99.1%	69.5	69.4	99.9%



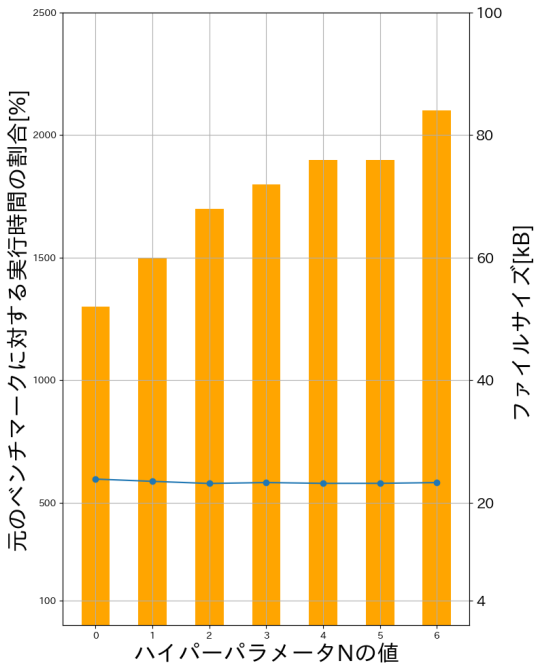
(a) List1



(b) Towers1



(c) Richards1



(d) Json1

図 4.3: ハイパーパラメータ N の値と実行時間およびコードサイズの関係

第 5 章

まとめと今後の課題

5.1 まとめ

本論文では, Promise オブジェクト以外を返す式に対して保守的に await を付与する場合を想定して, その際のオーバヘッドが await より小さい演算子 await?と, それを JavaScript 上で実現するためのコード変換を提案した. await?は標準の await と同じ動作をするが, Promise オブジェクト以外を返す式に付与された場合の速度低下を小さく抑える. Promise オブジェクトを返す式に付与された場合に実行するスローパスと, Promise オブジェクト以外を返す式に付与された場合に実行するファストパスを予め用意しておき, プログラムの実行時に適切なパスへ分岐する. ファストパスは実行時間が Promise オブジェクト以外を返す前提で最適化されており, 標準の await 相当の処理を省略することで実行時間を短縮している. また, 提案した手法に従い, await?を含むプログラムを純粋な JavaScript コードに変換する変換器を開発した.

await?の性能を評価するにあたって, Marr らの Cross-Language Compiler Benchmarking [1] に含まれるベンチマークを元に, 非同期処理を含むベンチマークを作成した. 非同期関数の割合を変化させることで, 元のベンチマークから複数のベンチマークを作成した. 実験から,Promise オブジェクト以外を返す式に付与された場合は実行時間が概ね短縮され, Promise オブジェクトを返す式に書かれた場合には同程度の性能を示すことを確認した. 実際の利用を想定し非同期関数の呼び出し回数の割合が 10% 程度のベンチマークに着目すると, 標準の await と比較して 13% 程度の速度改善がみられた. また, 継続の関数化を行うことによって, 実行時間の増加を最大で 1.5 倍程度のオーバヘッドが生じるが, 変換に伴うコードサイズの増加を半分以下に削減できることがわかった.

5.2　今後の課題

今後の課題として, await?を利用できる範囲の拡大が挙げられる. 本研究で提案したコード変換では, await?を含むループが存在する関数を変換の対象から除外している. しかし, イテレーションの実行順に依存関係が無い場合は, Promise.all() メソッドを使うことができると考えられる. また, 例外処理を含む関数にコード変換を適用した際の挙動については検証できていないため, 追加で確認を行う必要がある.

ベンチマークの改善についても述べる. 本研究では, await?の性能評価にあたり, 本来は非同期処理を行わないベンチマークを元に非同期処理を含むベンチマークを複数作成し, その実行時間を元に await?と await の性能評価を行った. ベンチマークの改善方法としては 2 つの方針が考えられる. 1 つは本研究で作成したベンチマークを改善すること, もう 1 つは新たなベンチマークを作成することである. 前者について, 例えば元のベンチマークからより多くのベンチマークを作成することが考えられる. 作成できるベンチマークがどの程度非同期関数を含むかは元のベンチマークによって大きく異なり, 特にマイクロベンチマークを元にしたものは 3 段階が限界であった. そのため, 実験から得られた実行時間のグラフにおけるプロットは一様ではなく, 評価を難しくしてしまった. 関数の呼び出し関係をより詳しく追うことで, 正常に動作するベンチマークをより多く作成し, 適した非同期関数の割合を持つものを選択して実験を行うことができると考える. 後者については, 実際に非同期なプロセスを挟むベンチマークの作成が挙げられる. 例えば通信にかかる時間を固定するといった対応が必要になると思われるが, 実際の利用シーンに近い環境を再現することで, より厳密な評価が得られることが期待される.

発表文献と研究活動

(1) 川向聡, 山崎徹郎, 千葉滋. 不要な場合でもオーバヘッドの小さい `await` の JavaScript 言語における実現に向けて. 日本ソフトウェア科学会 第 40 回大会. 2023.09.12-14.

参考文献

[1] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: Are we fast yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, p. 120–131. Association for Computing Machinery, 2016.

[2] D.P. Friedman and D.S. Wise. *The Impact of Applicative Programming on Multiprocessing.* Technical report (Indiana University, Bloomington. Computer Science Department). Indiana University, Computer Science Department, 1976.

[3] Ellen Arteca, Frank Tip, and Max Schäfer. Enabling additional parallelism in asynchronous javascript applications. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, Vol. 194 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[4] Ena Tominaga, Yoshitaka Arahori, and Katsuhiko Gondow. Awaitviz: a visualizer of javascript's async/await execution order. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, p. 2515–2524, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Thodoris Sotiropoulos and Benjamin Livshits. Static analysis for asynchronous javascript programs, 2019.

[6] Ohad Rau, Caleb Voss, and Vivek Sarkar. Linear Promises: Towards Safer Concurrent Programming. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, Vol. 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 13:1–13:27, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[7] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. Reasoning about the node.js event loop using async graphs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, p. 61–72. IEEE Press, 2019.

[8] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous javascript programs. *Proceedings of the ACM on Programming Languages*, Vol. 2, pp. 1–26, 10 2018.

[9] Alexi Turcotte, Michael D. Shah, Mark W. Aldrich, and Frank Tip. Drasync: Identifying and visualizing anti-patterns in asynchronous javascript. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, p. 774–785, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. *The Essence of JavaScript*, p. 126–150. Springer Berlin Heidelberg, 2010.

[11] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

[12] Gerald Sussman and Guy Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, Vol. 11, pp. 405–439, 12 1998.

謝辞

本研究を進めるにあたり, ご指導をいただいた本学千葉滋教授, 本学山崎徹郎特任助教に心から感謝いたします. 毎週のミーティングでは的確なアドバイスをいただき, 論文執筆や発表練習などについてもご助力を賜りました.

また, 本学鵜川始陽准教授, 本学千葉研究室の先輩方, 同期には, 研究生活の様々な場面で相談に乗っていただきました. 本当にありがとうございました.