平成26年度 修士論文

オフラインWebアプリケーションにおける事前データ取得の半自動化

電気通信大学 大学院情報システム学研究科
情報システム基盤学専攻
1353001　磯谷 俊明

主任指導教員　小宮 常康　准教授
指導教員　多田 好克　教授
指導教員　本多 弘樹　教授

提出日　平成27年2月10日

目次

第 1 章　はじめに　3

第 2 章　背景　5
- 2.1　Web アプリケーションを取り巻く環境 5
- 2.2　Web アプリケーションの実行形態の違い 6
- 2.3　Web アプリケーションの記述面の違い 10
- 2.4　本研究の目的 12

第 3 章　関連技術・研究　14
- 3.1　関連技術 14
- 3.2　関連研究 15

第 4 章　提案手法の設計　20
- 4.1　local Storage へのアクセスとサーバへのアクセスを隠蔽する処理 20
- 4.2　コールバック関数を含まない記述スタイル 21
- 4.3　プリフェッチコードの自動挿入機能 22
- 4.4　CPS 変換によるコールバック関数の自動受け渡し 23
- 4.5　アプリケーション実行までの流れ 26

第 5 章　提案手法の実装　27
- 5.1　プリフェッチコードの自動挿入器 27
- 5.2　CPS 変換 29
- 5.3　JavaScript で記述された Scheme インタプリタ 29

第 6 章　今後の課題　34

第 7 章　おわりに　35

図目次

2.1 従来の Web アプリケーション 7
2.2 完全オフライン型の Web アプリケーション 8
2.3 準オフライン型の Web アプリケーション 9

3.1 Application Market のスマートフォン版の画面 (文献 [7] より抜粋) . . . 16
3.2 Application Market のパソコン版の画面 (文献 [7] より抜粋) 16
3.3 オーサリング使用前の画面 (文献 [7] より抜粋) 17
3.4 オーサリングツール使用後の画面 (文献 [7] より抜粋) 17
3.5 WOPRE の使用イメージ図 18

5.1 プリフェッチコード自動挿入の概略 28

第 1 章

はじめに

近年 Gmail や Google Maps などの Web アプリケーションが多くのユーザにより利用されている. Web アプリケーションは使用する実行環境に依存しないという特徴がある. しかし Web アプリケーションのダウンロード時だけでなく, その実行の間中, ネットワーク接続がオンライン状態でなければならない事も多い. 近年ではスマートフォンから Web アプリケーションを使用するケースも多く, オフライン状態下で使用できない事は不便さにつながる可能性がある. その点を解消したものがオフライン Web アプリケーションである. オフライン Web アプリケーションはオフライン環境下でも実行が可能な Web アプリケーションの総称である. オフライン Web アプリケーションはアプリケーションの実行に必要なデータをあらかじめ Web サーバから取得する事でオフライン状態での実行を可能にしている. オフライン Web アプリケーションは Web サーバとの通信回数が従来の Web アプリケーションと比べて少ないので, スマートフォン上などで利用した場合はバッテリーの寿命が延びる, アプリケーションのレスポンスが向上できるなどのメリットもあり得る.

オフライン Web アプリケーションにおいては Web サーバから Web ブラウザ側に実行に必要なデータを一括で事前取得 (以下プリフェッチ) する事が基本である. 従ってプログラム中のプリフェッチコードは最初のデータ取得時の記述のみでよく, 通信処理に伴うコールバック関数の記述コストは少ない.

しかしデータの一括取得が困難な場合がある. この問題は従来では一括取得後は常にオフライン状態である事を想定していたオフライン Web アプリケーションにおいて, データの一括取得後も必要に応じて一時的にオンライン状態になる事を想定したオンデマンドプリフェッチを採用する事で解決できる. ただしオンラインになるタイミングは選択できないものとする. これは電波が不安定になる場所での使用を想定している為である. このようにデータの一括取得後に, 一時的にオンライン状態になることを想定するアプリケーションを準オフライン型, また一括取得後のデータ通信を行わないアプリケーションを完全オフライン型と便宜上呼ぶことにする.

準オフライン型においてはオンラインになるタイミングを選べないため必要なデータを予測し, オンライン時にプリフェッチをする事が考えられる. しかしこの方法を採用する

と実行に必要なデータをプリフェッチするタイミングを考慮しつつプログラミングを行うことが求められるため, データのプリフェッチを行うコードを何度も修正する事が予想される. 例えばオフライン時にプリフェッチを行っても失敗する事からプリフェッチコードの実行を試みる間隔を変更する事などが考えられる. さらにプリフェッチコードに伴うコールバック関数の書き換えも行う必要があり非常に煩わしい.

そこで本研究では準オフライン型の Web アプリケーションにおいて, プリフェッチコードをプログラムのロジックとは別に用意した情報から自動生成し, プログラム中に自動挿入する手法を提案する. またコールバック関数などの煩雑な処理を記述する必要のない手法についても提案する. 本研究の成果によりオフライン Web アプリケーションにおけるプログラムの複雑さや修正コストが軽減する事が期待できる.

第 2 章

背景

2.1 Web アプリケーションを取り巻く環境

近年 Gmail や Google Maps などの Web アプリケーションが多くのユーザにより利用されている． Web アプリケーションは通常のデスクトップアプリケーションと比較して, アプリケーションを実行させる端末によって開発に用いるプログラミング言語を使い分ける必要がないという特徴を持つ． 例えば通常 Android 上で動作するアプリケーションを開発する際には Java を用いる事が普通である. Android 上で開発したアプリケーションと同様のアプリケーションを iOS 上で動作させる場合, 新たに Objective-C というプログラミング言語を用いて開発し直す必要がある. すなわち同じアプリケーションを異なる端末で実現しようとするときに実行を行う端末によってプログラミング言語を使い分ける必要がある. こういった事は Web アプリケーションでは起こらない.

また Web アプリケーションはオフライン時には使用する事は出来ないという特徴も持ち合わせている． これは Web アプリケーションでは実行に必要なデータを Web サーバから取得するため, オフライン状態では必要なデータを取得する事ができない事による． 近年ではスマートフォンから Web アプリケーションを利用する機会も増えている事から, オフライン状態で使用できない事は不便になってしまう事が考えられる． なぜならばスマートフォンではパソコンと異なり, ネットワークが不安定な場所で使用する事も想定できるためである． この点を改善する方法としてオフライン Web アプリケーションを導入するという方法がある.

オフライン Web アプリケーションとは, オフライン状態においても Web アプリケーションの実行を可能にする Web アプリケーションの実行形態である. アプリケーションの実行に必要なデータをあらかじめ Web サーバから事前取得 (以下プリフェッチ) し, オフライン時にはそのデータを実行に使用する事でオフライン状態での実行を可能にしている． オフライン Web アプリケーションにおいてはデータの事前取得は一括で行う事が基本である． 一括取得を行う事によりその後はインターネット接続を一切せずとも, アプリケーションの実行が可能になる.

しかしながら, 一括でデータを取得できないケースも存在する． 例えば地図アプリケー

ションにおいて世界中の地図画像を事前取得する事は現実的ではない．そこでこのような場合に, あらかじめ実行に必要であると思われるデータを予測し, どこかオンラインになったタイミングでデータを取得する事を考える．これはオフライン Web アプリケーションにおいて, 一括取得後はインターネット接続する事を考慮に入れないという今までの前提を崩し, 一括取得後のどこかのタイミングでオンライン状態になる事を想定し, そのタイミングで実行に必要なデータを取得できるようにする事を考える事を意味する. また必ず一括取得後にオンライン状態になるとは限らないので, その時の事も考慮に入れる事が必要である.

本研究においては便宜上, 従来の一括取得後にはインターネット接続を一切行う事を考慮しないオフライン Web アプリケーションを完全オフライン型の Web アプリケーション, そして一括取得後にどこかのタイミングでオンラインになる事を考慮に入れるオフライン Web アプリケーションを準オフライン型の Web アプリケーションと呼ぶ事にする.

2.2 Web アプリケーションの実行形態の違い

従来型の Web アプリケーションと完全オフライン型の Web アプリケーション, 準オフライン型のオフライン Web アプリケーションでは実行形態が異なる．そこで従来型の Web アプリケーションと完全オフライン型の Web アプリケーションおよび, 準オフライン型の Web アプリケーションの実行形態をそれぞれ図 2.1, 図 2.2, 図 2.3 に示す．Web サーバおよび Web ブラウザから伸びている線において, 実線はオンライン状態, 波線はオフライン状態である事を表す.

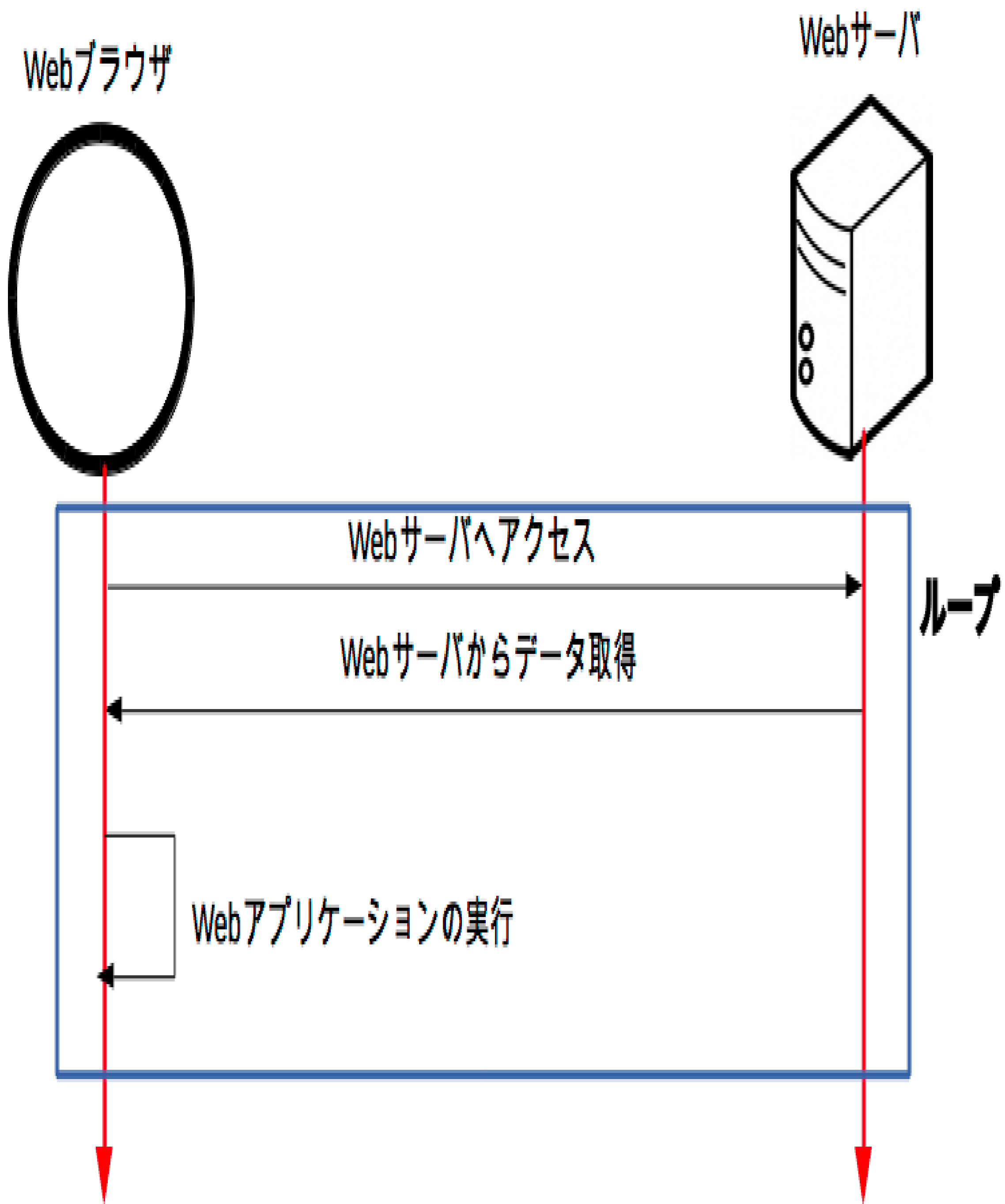

図 2.1　従来の Web アプリケーション

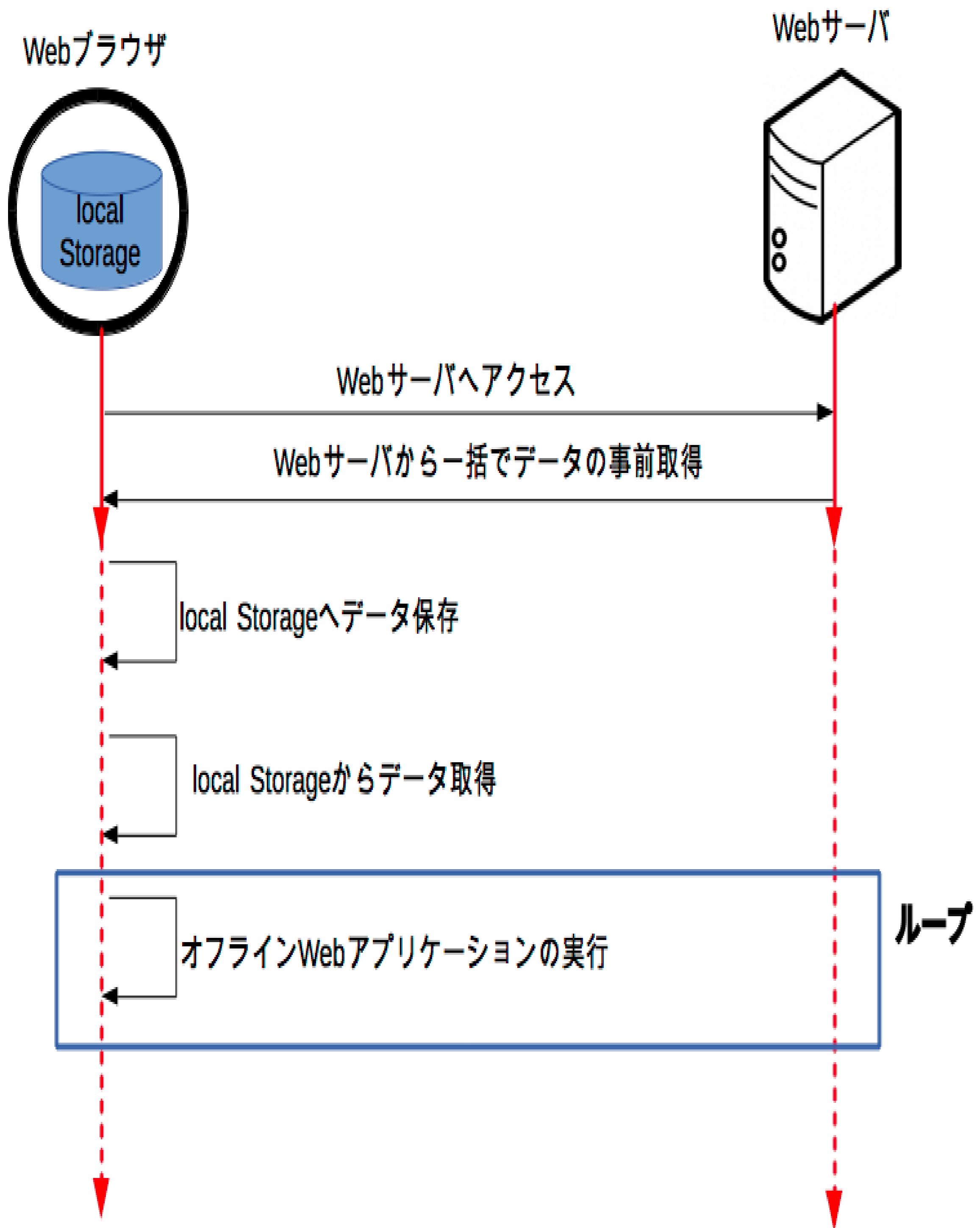

図 2.2 完全オフライン型の Web アプリケーション

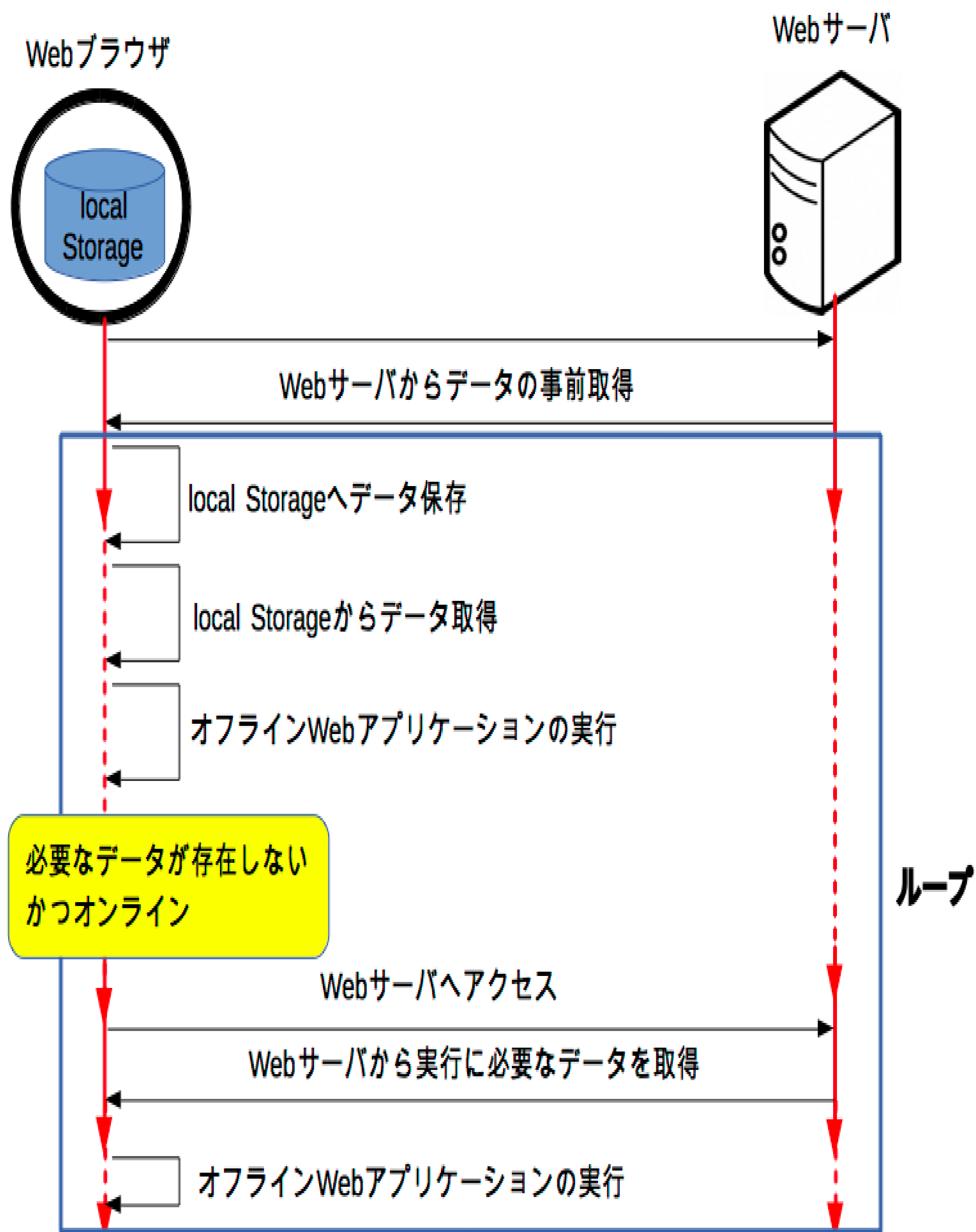

図 2.3 準オフライン型の Web アプリケーション

図 2.1 は常にオンライン状態を想定しており，任意のタイミングで実行に Web サーバへアクセスし，データを取得し Web アプリケーションを実行する．

図 2.2 ではまずオンライン状態の時にアプリケーションの実行に必要なデータを Web サーバから一括で事前取得する．その後，取得したデータを local Storage に保存する．そ

してアプリケーションを実行するときに local Storage から実行に必要なデータを取り出しアプリケーションの実行を行う.

図 2.3 ではまずオンライン状態の時に Web サーバからアプリケーションの実行に必要なデータの事前取得を行う. その後取得したデータを local Storage に保存する. そしてアプリケーションを実行するときに local Storage から実行に必要なデータを取り出しアプリケーションの実行を行う. アプリケーションの実行に必要なデータを予測し, もし必要なデータが local Storage 内に存在しないかつオンライン状態であれば, 必要なデータを事前取得し local Storage に保存する. その後 local Storage 内のデータを使用し, アプリケーションの実行を行う. 必要となるデータが local Storage 内に存在しない事がわかってもオフライン時の場合は, データを取得する事は出来ないのでその時はエラー処理をするなどして適切な対応を行うようにする. また必要なデータの予測に失敗した時もエラー処理などの適切な対応を行うようにする. 以上の事を準オフライン型の Web アプリケーションは行う.

2.3 Web アプリケーションの記述面の違い

従来の Web アプリケーションにおけるクライアントサイドの処理の記述を行うためには JavaScript を用いる事が通例である. しかし JavaScript はシングルスレッドのため, Web サーバへ通信を行うと Web サーバからのレスポンスが返ってくるまで, Web ブラウザが固まってしまい, 一切のマウス操作を受け付けなくなるという問題がある. この問題を回避するために Ajax という技術がある.

Ajax では従来は同期通信で通信を行っていたものを, 非同期通信で通信を行う. つまり Web ブラウザから何らかのリクエストを行い, Web サーバからのレスポンスが来るまで待つという従来の方法を, Web サーバからのレスポンスが返ってくるまで待たずに次の処理を開始する. これにより Web サーバへの通信後も, 一切待たずにマウス操作などを行う事が出来るようになる. しかし非同期処理の場合は, Web サーバからレスポンスが返ってきた後の処理をコールバック関数として記述する必要がある. これは Web サーバからのレスポンスを使用しようとすると, 既にスレッドが終わっているのでそのデータを使用する事が出来ないためである.

コールバック関数はプログラムの流れが変わってしまうため, プログラムの可読性を下げてしまう可能性がある. またコールバック関数が大量にネストした場合などは非常に修正が面倒である. この問題はコールバック地獄として知られている.

また完全オフライン型の Web アプリケーションにおいては基本的に通信処理は最初の事前取得時のみでよく, その後の通信処理は不要である. コールバック関数はその通信に伴うもののみ記述すればよい. そのためコールバック関数によって大きくプログラムの

流れが乱されたり, コールバック関数がネストする事は考えにくい.

準オフライン型の Web アプリケーションにおいては一括取得時以外にもプリフェッチコードを記述する可能性がある. また準オフライン型の Web アプリケーションではオンラインになるタイミングがわからないので, プログラム中のプリフェッチ位置を変える可能性がある. 例えばオフライン時にプリフェッチを行っても失敗する事は明らかなので, きちんとオンライン時にプリフェッチが行われるようにプログラムを修正する事が考えられる. またプリフェッチは非同期の通信処理なのでコールバック関数もプリフェッチコードとあわせて記述する必要がある. プログラム中のプリフェッチ位置を変更するたびにコールバック関数もあわせて修正するのは大変煩わしい.

例としてソースコード 2.1 に 3 つの値を加算する処理を従来の Web アプリケーション (オンライン型) の記述方法で記述したプログラムを示す. また同様の処理を完全オフライン型で記述したプログラムをソースコード 2.2 に, 準オフライン型で記述したプログラムをソースコード 2.3 に示す.

ソースコード 2.1 3 つの値の加算処理 (オンライン型における記述)

```
1 function proc(){//プログラム本体
2     xhr('a.txt',funciton(a_val){
3         xhr('b.txt',funciton(b_val){
4             xhr('c.txt',funciton(c_val){
5                 print(a_val+b_val+c_val);
6             });
7         });
8     });
9 }
```

ソースコード 2.2 3 つの値の加算処理 (完全オフライン型における記述)

```
1 function init(){//初期化処理
2     prefetch('a.txt','b.txt','c.txt');
3 }
4
5 //プログラム本体
6 function proc(){
7     print(localStorage_get(a_key)+localStorage_get(b_key)+localStorage_get(c_key));
8 }
```

ソースコード 2.3 3 つの値の加算処理 (準オフライン型における記述)

```
1 function init(){//初期化
2     prefetch('a.txt','b.txt');
3 }
4
5 function proc(){//プログラム本体
6     get('a_key', 'a.txt', funciton(a_val){
```

```
 7            get('b_key', 'b.txt', funciton(b_val){
 8                get('c_key', 'c.txt', funciton(c_val){
 9                    print(a_val + b_val + c_val);
10                });
11            });
12        });
13    }
```

ソースコード 2.1 において xhr は, Web サーバとの通信を行いデータを取得する関数である. 第 1 引数として取得するデータの URL, 第 2 引数としてデータ取得後の処理をコールバック関数として与える. コールバック関数中にも通信が発生するのでさらにコールバック関数を記述し, 5 行目でサーバから取得したデータを用いて加算処理を行い表示をしている.

ソースコード 2.2 においては, 関数 localStorage_get は local Storage に保存されている値をキーを用いて取り出す関数である. 引数としてキーを指定する. まず始めに初期化のための関数 init が呼び出される. これは Web ページの読み込み後すぐに呼び出される関数である. 初期化処理によって実行に必要なデータのプリフェッチが行われる. その後プログラム本体が呼び出されると, local Storage 内のデータを用いて加算を行い, その後表示する.

ソースコード 2.3 において, 関数 get は実行に必要なデータが local Storage 中に存在すれば第 1 引数として指定されているキーを用いてデータを取得する, もしデータが local Storage 内に存在せずかつオンライン状態であれば, サーバへアクセスしデータを取得する. get は通信を行う可能性があるためコールバック関数を第 3 引数として指定する必要がある. なお本来は, オフラインであるためにサーバへアクセスできない場合は, 適切なエラー処理などを行い, アプリケーションの実行に必要な状態の一貫性を保つ処理記述が必要である.

まず初期化のための関数 init が呼び出される. これはソースコード 2.2 と同様である. その後プログラム本体が呼び出される. その中で表示処理を行うが, その表示処理の中でコールバック関数がネストした get 関数を記述する必要がある. 完全オフライン型のコードと比べるとプログラムの可読性が落ち, プログラム修正も面倒になっている事が分かる.

2.4 本研究の目的

本研究では準オフライン型の Web アプリケーションにおけるプログラム開発・修正 (データ取得処理に伴うコールバック関数を含む) のやりやすさを改善する事を目的とする. 具体的には, 完全オフライン型アプリケーションと同様の記述で準オフライン型アプ

リケーションを実現する機能を提案する．本研究の成果により準オフライン型の Web アプリケーション開発の際のプログラム修正および開発コストが減少する事が期待できる．

第 3 章

関連技術・研究

3.1 関連技術

3.1.1 HTML5

HTML5[1] はオフライン Web アプリケーション開発のための要素技術をいくつか持っている.

local Storage は Web ブラウザに搭載されている, データを永続化できるストレージである. local Storage はキー・バリューのデータ構造を持っておりキーを使って値にアクセスする事が出来る. また値を保存したい場合はキーと値を HTML5 において使用できる API(localStorage. setItem) を使用する事で local Storage 上に値を保存する事が可能である. local Storage から値を取得したい場合は専用の API(localStorage. getItem) が用意されている.

オフライン Web アプリケーションにおいては, local Storage をサーバから取得してきたデータを保管する場所として扱う.

本研究においても, 主にサーバから取得した実行に必要なデータを保存するためのストレージとして local Storage を利用する.

またキャッシュマニフェストはデータをあらかじめキャッシュしておくための仕組みである. マニフェストファイルというファイルにキャッシュしておきたいファイルのファイル名を指定する事でそのファイルをキャッシュする事ができる. またキャッシュしてほしくないファイル, つまり Web サーバへ直接アクセスして利用したいファイルを指定する事が可能である. 本研究ではマニフェストファイルは使用しない.

3.1.2 Google Gears

Google Gears[4] は Google 社でかつて提供されていた Web アプリケーションをオフライン対応化させる事ができる, Web ブラウザの拡張機能である. しかしながら現在は開発が終了している. Google Gears の多くの機能は HTML5 に取り込まれている.

Google Gears の長所としては導入できる Web ブラウザに制限がないという事であ

る． また Google Gears の短所としては Google スプレッドシートなどの Google 社から提供されているアプリケーションのみ対象にしている点である． そのためそれ以外のアプリケーションに Google Gears を適用する事ができない. また記述性に関しては特に言及されていない.

3.1.3 オフライン・ファースト

オフライン・ファースト [2] はオフライン Web アプリケーション開発を行う際の開発の指針である． オフライン・ファーストでは従来はオンラインが前提で開発されていたオフライン Web アプリケーションをオフラインを前提に開発するという事を提言している.

従来ではオフライン Web アプリケーションを開発する際にはデータの場所としてサーバ側とクライアント側という 2 つを仮定していた． オフライン・ファーストの場合は Web アプリケーションの実行に必要なデータは全てクライアント側にあると仮定している． これにより特にサーバ側のデータを意識する必要がなくなるというメリットがある． このため特にプログラマはサーバ側のデータに気を配る事なくクライアント側のデータの管理に専念してプログラミングを行う事ができる． またオフライン Web アプリケーションを実現するためにサーバから移送してきたデータをサーバとの通信コードと合わせてカプセル化する事で, プログラムの見通しのよさを改善する事ができる方法が示されている.

3.2 関連研究

3.2.1 オフライン Web アプリケーションの実行を可能にするランタイム環境

Yung ら [7] はオフライン Web アプリケーションの実行を可能にするランタイム環境 WOPRE を提案している． WOPRE は開発者が開発した Web アプリケーションを WOPRE のサーバにアップロードする事でオフライン対応化する事が可能で, Google Gears のように特定のアプリケーションでのみ使用可能という制限はない． ユーザはサーバからオフライン対応化された Web アプリケーションを自身の端末にダウンロードし, 使用する事ができる. これは Apple の App Store によく似た構造を持っている. WOPRE においては Application Market が App Store の役割を担っている． Application Market はスマートフォンからアクセスした場合と, パソコンからアクセスした場合とでは外観が異なる． 図 3.1 にスマートフォン版の外観を, 図 3.2 にパソコン版の外観をそれぞれ示す.

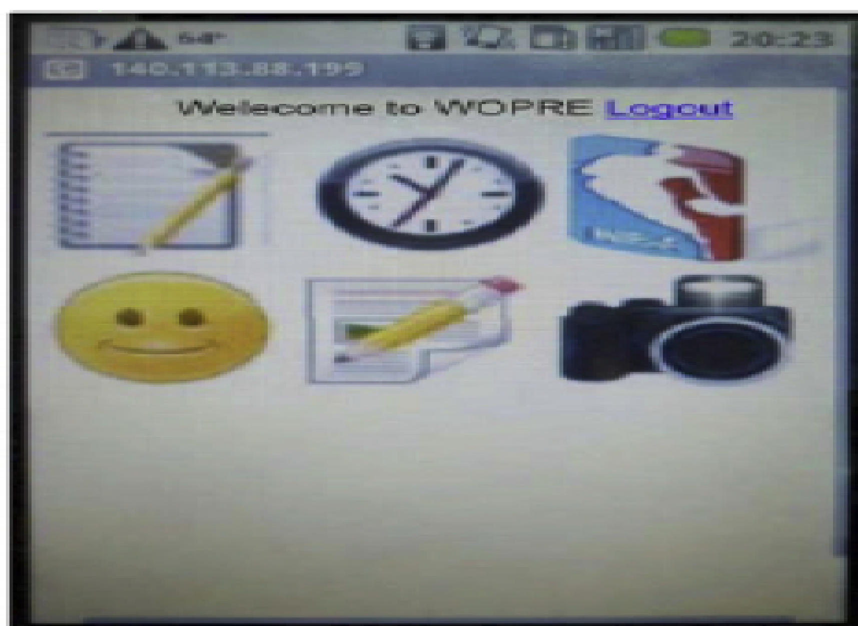

図 3.1　Application Market のスマートフォン版の画面 (文献 [7] より抜粋)

Welecome to WOPRE Logout

id	name	icon	sutible version		
1	clock		Chrome Lite		
2	NBA		Chrome Lite	Safari	
3	world_map		Chrome Lite		Mobile IE
4	ToDoList		Chrome Lite	Safari	Mobile IE
5	game		Chrome Lite		
6	mail		Chrome Lite	Safari	Mobile IE
7	money		Chrome Lite	Safari	Mobile IE
8	photo		Chrome Lite	Safari	

図 3.2　Application Market のパソコン版の画面 (文献 [7] より抜粋)

また WOPRE では Web ブラウザによって表示形式が異なる問題も解決している．Web アプリケーションを使用していると, Web ブラウザによって表示が若干異なるという事がある． WOPRE ではユーザがアプリケーションをダウンロードするときに Web ブラウザに適した形に変更するためのオーサリングツールをダウンロードする事ができる． こ

のオーサリングツールを使用する事により Web ブラウザに適した形に成形するという仕組みになっている．オーサリングを使用した前後をそれぞれ図 3.3, 3.4 に示す．

図 3.3　オーサリング使用前の画面 (文献 [7] より抜粋)

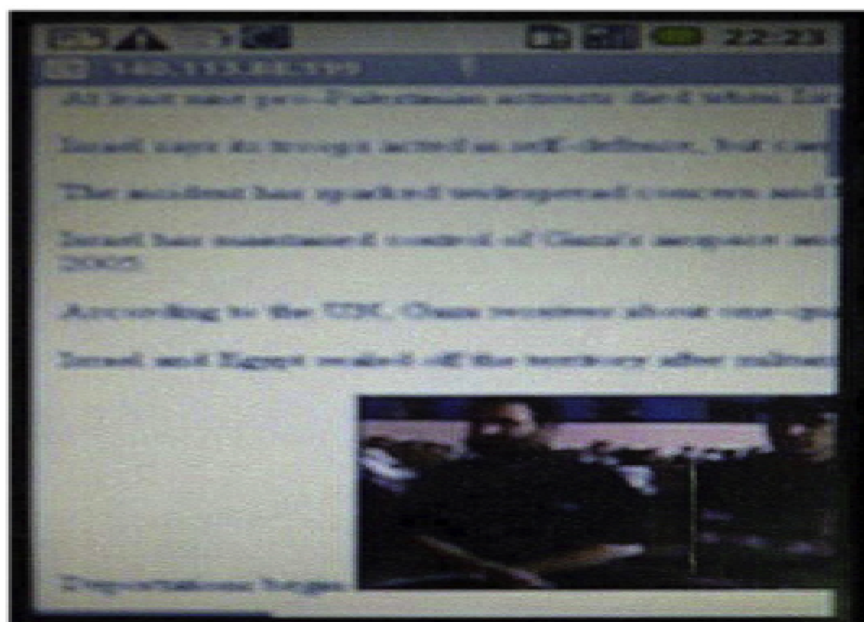

図 3.4　オーサリングツール使用後の画面 (文献 [7] より抜粋)

しかしながらプログラムの記述性に関しては言及されていない．最後に WOPRE の使用イメージを図 3.5 に示す．

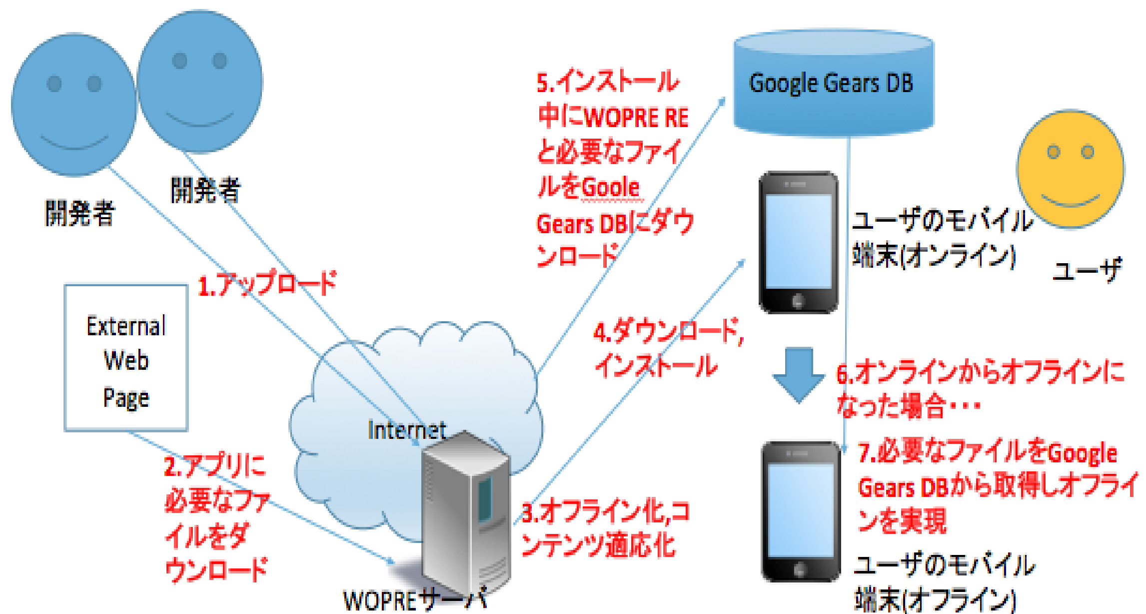

図 3.5　WOPRE の使用イメージ図

3.2.2　オフライン環境で動作する分散システム向けのプログラミングモデル

Yaron ら [8] はオフライン環境で動作する分散システム向けのプログラミングモデルを提案している. このモデルではアプリケーションをオフライン化するために, アプリケーションの構成要素をローカル環境に全てコピーする方法を採用している. そのためサーバ側に存在するコピー元のアプリケーション, またはクライアント側に存在するコピー先のアプリケーションのどちらかに変更が加えられた場合は両者の整合性がとれなくなるが, 定期的に確認する事によりできる限り衝突を避ける工夫もなされている.

しかしながらこのモデルでは移送可能な分散オブジェクトとメッセージによる分散コンピューティングモデルを用いているため既存の Web 技術とのモデルのギャップが大きいと言う特徴がある. 従ってそのまま Web アプリケーションの世界に適用する事は困難である.

3.2.3　JavaScript の初期読み込み時間を高速化するツール

Benjamin ら [9] は Web サーバから Web ブラウザへの最初の JavaScript の読み込みを高速化するツール Doloto を提案している. Doloto では JavaScript の関数呼び出し部分と関数定義部分を区別し, まず最初に関数呼び出し部分を Web ブラウザに対して読み込んでいる. 関数定義部分は必要に応じて（その関数が呼び出される前に）ダウンロード

する手法を採用している．これにより一番はじめの JavaScript の読み込み時間は削減する事ができる．また関数定義部分においてもコードの最適化を行いできる限りコード量を削減する工夫がなされている．

コード量の削減と JavaScript の一番始めの読み込みの高速化を目的とした研究であるため, プログラムの記述性に関しては特に言及されていない．本研究との類似点は実行に必要なデータを一括で取得する事はせずに必要に応じてプリフェッチを行う点である．また Doloto はこの論文を元にしたツール [3] が実際に Microsoft 社から提供されており, そのツールを実際に使用する事ができるようになっている．

3.2.4 非同期に発生したイベントを検出するアルゴリズム

非同期に発生したイベントを検出するためのアルゴリズムとして Feeley[10] による Balanced Polling がある．Balanced Polling では定期的にイベント発生を検査（ポーリング）するコードをプログラム中に挿入する事でイベントの発生を検出できるようにしている．基本的にはプログラムの制御フローを上から順番に見ていき, 直前のポーリングを行った後に実行する命令数をカウンタを用いて数える．命令数がある一定の値に達する直前にポーリングのためのコードを挿入する. これにより定期的にポーリングを行うコードを挿入する仕組みになっている．

しかし Balanced Polling ではコンパイル時にポーリングが行われる場所が全て決まるために, ループや関数呼び出しの前後において過剰にポーリングを行うコードが挿入されてしまう可能性があるという問題がある. この問題を解決する方法として田中ら [13] は Punctual Pollling を提案している．Punctual Polling では直前のポーリングから実行する命令数の上限に加えて下限を設定する事でポーリングを行う回数を上限と下限の間に押さえる事を行う．

例えばループの場合を考える．ループにおいてはポーリングを行うコードを必ず一つは挿入する事が必要だが, ループの本体が小さい場合はポーリングの間隔が狭くなってしまう．その結果オーバーヘッドが増加する事が起こると考えられる．そのため前のポーリングから実行される命令数の下限を設定する事で, 本体が小さいループなどにはポーリングを行うためのコードを挿入しないようにする．これによりオーバヘッドを削減する事が期待できる．

本研究との類似点はあるコード（本研究においてはプリフェッチコード, Balanced Polling や Punctual Polling においてはポーリングを行うコード）を挿入する場所を調整する点にある．

第 4 章

提案手法の設計

本研究においては準オフライン型の Web アプリケーションにおけるプログラムの煩雑さや修正コストを軽減する事を目的としている. そのために get 関数, コールバック関数を伴わない記述手法, プリフェッチコードの自動挿入の導入を行う. 本章ではそのため設計を示す.

Web アプリケーションの記述には通常 JavaScript を用いる. しかし本研究においては便宜上 Scheme を用いる事にする.

4.1 local Storage へのアクセスとサーバへのアクセスを隠蔽する処理

準オフライン型 Web アプリケーションのナイーブな記述ではサーバとのデータ通信を行うコード (サーバへのアクセス成功時に実行されるコード) と local Storage へのデータアクセスを行うコード（サーバへのアクセス失敗時に実行されるコード）を併記する必要があった. 本研究においては local Storage へのアクセスとサーバへのアクセスを融合して一つの API とした, get という組み込み関数を導入した.

get はオフライン時または local Storage に必要なデータあるときには local Storage へのアクセス機能として働く. local Storage に必要なデータが存在せず, かつオンライン時にはサーバへのアクセス機能として働く. すなわち get におけるサーバへのアクセスは local Storage に必要なデータがない場合の保険のようなものであると考える事ができる. get の実装については後述する.

get の導入により local Storage へのアクセスと同等の記述で暗にサーバへのアクセスを行う事ができ, プログラムの複雑さを軽減する事が可能である. しかしながら get のナイーブな実装では, get 以降の処理をコールバック関数として表現し, それを get の引数として渡す必要がある. また get は local Storage へのアクセッサ関数の代わりとして使うことを意図するので一般に, 複数の文で記述される通信コードに比べて, プログラムのあらゆるところで頻繁に使用され得る. したがって, そのそれぞれの使用場所においてコールバック関数を用意する煩わしさが重大な問題となる. この問題の解決策については

後述する.

4.2 コールバック関数を含まない記述スタイル

本研究では get の使用についてコールバック関数を含まない記述スタイルを採用し, それを実現するための手法を提案する. これは get のナイーブな実装で問題となるコールバック関数が大量に必要となる問題を回避するためである.

コールバック関数はプログラムの処理の流れを字面上分断するのでプログラムの可読性が低下する, プログラムの保守性が下がるという問題がある. 本研究の手法を採用する事によりプログラムの可読性の低下およびプログラムの保守性の低下の問題も合わせて解決できる.

例えばソースコード 4.1 のような Scheme プログラムを考える. セミコロン以下はコメントを表す.

ソースコード 4.1 コールバック関数を伴う加算プログラム

```
1 (get
2   (lambda (x);k1
3         (get
4          (lambda (y);k2
5                 (print (+ x y)))
6               url2 key2))
7      url1 key1)
```

これはサーバもしくは local Storage から 2 つのデータを取得し, その結果を加算し表示するプログラムである. 本来であればソースコード 4.2 のように記述したいが, 既存のブラウザにおける JavaScript におけるナイーブな get の実装ではこのようになる. k1,k2 はコールバック関数, url1 と url2 はサーバへのアクセス時に取得したいデータの場所を表す URL, key は local Storage にアクセス時に値にアクセスするためのキーである. この get は URL もしくはキーを通して該当データを取得した後, 第 1 引数に与えられたコールバック関数を呼び出すことで後続の処理を再開する.

例えばソースコード 4.1 の加算の引数にもう一つ url3/key3 からのデータ z を加えようとすると, コールバック関数 k2 を（k1 のように）書き直す必要があるが, 4.1 に対して同じ修正を加える（単に (get url3 key3) を + の引数に追加）のに比べて直感的ではない. またソースコード 4.1 の 1 行目の直前にプリフェッチコードを挿入すると, プリフェッチコードもコールバックを伴うので, 1〜7 行目のコードはそのコールバック関数の本体へ移す必要がある, 同様にもし 1 行目の get の実行直後にプリフェッチコードを実行させたいのであれば 3〜6 行目をそのコールバック関数の本体へ移す必要がある. このように get を含む式を変更したり, 途中にプリフェッチコードを挿入するとコールバック関数が絡ん

だ修正が必要となり大変煩わしい.

この問題を解決するために本研究においてはコールバック関数を伴わない記述スタイルを提供している. 例えばソースコード 4.1 のプログラムを提案手法の記述方法で書くとソースコード 4.2 のようになる.

ソースコード 4.2 コールバック関数を伴わない加算プログラム

```
1 (+ (get url1 key1) (get url2 key2))
```

ソースコード 4.2 はコールバック関数を伴わない加算プログラムである. get はサーバもしくは local Storage へのアクセスを行う関数, url1 および url2 はサーバへのアクセス時に取得したいデータの場所を表す URL, key1 と key2 は local Storage にアクセス時に値にアクセスするためのキーである.

ソースコード 4.2 ではソースコード 4.1 と比べると複雑さが改善されている. またプリフェッチコードを挿入してもコールバック関数を記述する必要がないのでプログラムの修正が非常にやりやすい. また先述した get の導入によってコールバック関数の修正コストが増加するという問題も, コールバック関数を記述する必要がなくなる事で解決出来る.

4.3 プリフェッチコードの自動挿入機能

準オフライン型においてはオンラインになるタイミングを選ぶ事はできない. そのためプリフェッチコードの適切な場所は必ずしも自明ではなく, プリフェッチコードを適切な場所に挿入するための試行錯誤が必要である. 本研究においてはプリフェッチコードをプログラムのロジックとは別に記述された情報を基にプリフェッチコードを自動挿入する手法を提案する. またプログラムのロジックと分離する事でプログラムの修正をやりやすくするという効果が期待できる.

4.3.1 自動挿入器に与える情報の記述方法

プリフェッチコードの自動挿入に必要となる情報はソースコード 4.3 のように記述する

ソースコード 4.3 プリフェッチコードの自動挿入器が必要とする情報

```
1 (prefetch−when <function−name>)
```

ソースコード 4.3 において prefetch-when はキーワード, <function-name>は関数名を表す. また今回は Scheme においてリスト形式が取り扱いやすいという理由からリスト形式での記述を採用している.

4.3.2 プリフェッチコードの自動挿入器がプリフェッチコードを入れるタイミング

コード挿入器は<function-name>で指定された関数名の関数呼び出し式の直後にプリフェッチコードの自動挿入を行う．これはある関数が呼び出された事により local Storage 内のデータが消費されたと考えられるため, 消費されたデータの穴埋めをする役割を持たせるためである．

例えば地図アプリケーションにおいては, 地図の描画により local Storage 内のデータを消費したと考えられるので, 地図の描画を行う関数の関数呼び出し式の直後にプリフェッチコードを挿入する．

4.4 CPS 変換によるコールバック関数の自動受け渡し

本研究においては Scheme を用いたコールバック関数を伴わない記述スタイルを採用している．また CPS(continuation passing style) 変換を行いコールバック関数相当の後続の処理を関数として自動的に生成し, 次に呼び出される関数の引数として自動的に渡す事を行う．

CPS(継続渡しスタイル) とはプログラムの制御を継続を用いて陽に表すプログラミングスタイルの事を表す．通常のプログラミングスタイル (ダイレクトスタイル) では計算の途中 (例えば引数の評価の順序など) は暗黙になっている場合があるが, CPS にする事によりそれらが全て陽に表される．また CPS においてはダイレクトスタイルのように値を返す代わりに, 継続を引数として受け取り, その継続に計算結果を渡す．CPS 変換とはダイレクトスタイルのプログラムを CPS のプログラムに変換する事である．

CPS は今回のように継続を自動的に次に呼び出される関数の引数として渡す以外に, コンパイラ作成の用途として用いられる事がある．CPS をコンパイラ作成の用途に用いた論文として Guy L. Steele, Jr. による RABBIT[12], Kranz らによる ORBIT[11] などが知られている．

4.4.1 CPS 変換器が対応できる特殊形式

今回用意した CPS 変換器は関数呼び出し式およびいくつかの特殊形式に対応している．　対応している特殊形式は以下の通りである．

- (lambda (<パラメータ>...) <式>...)
- (begin <式>...)
- (if <式> <式> [<式>])

- (quote <データ>)

特殊形式 lambda は関数を作成する時に使用する. 特殊形式 begin は, 与えられた式を順番に評価し, 最後に評価した結果を返す. 特殊形式 if は条件式を記述するときに使用する. 特殊形式クォートは<データ>を式ではなくリテラルデータとして使用したい時に使用する.

また関数呼び出し式, および上述の特殊形式以外には対応していないので, それ以外の特殊形式がプログラム中に現れると適切な変換を行う事ができない.

4.4.2 CPS 変換の例

以下のようなプログラムを考える.

ソースコード 4.4 CPS 変換前の Scheme プログラム

```
1 (+ (+ (get data1 a) (get data2 b)) (get data3 c))
```

このプログラムはサーバもしくは local Storage から取得した 3 つの値を加算するプログラムである. data1, data2, data3 はそれぞれサーバアクセス時に必要となる URL, a, b, c は local Storage アクセス時に必要となるキーを表すとする.

ソースコード 4.4 のプログラムを CPS 変換するとソースコード 4.5 のようなプログラムになる. このコード中の lambda 式は継続と呼ばれる, ある時点以降の残りの計算を表現している. 例えば, 3 行目の lambda 式は (get G19 G20)（3 行目の get 及び 16 行目の引数からなる関数呼び出し式）の計算後の残りの計算を表している. 変数 G11 はその計算結果（get の返値）を受け取る. この継続は get 実行後に呼び出すべきコールバック関数に相当する.

ソースコード 4.5 CPS 変換後の Scheme プログラム

```
1  ((lambda (G19)
2     ((lambda (G20)
3        (get (lambda (G11)
4               ((lambda (G14)
5                  ((lambda (G15)
6                     (get (lambda (G12)
7                            (+ (lambda (G3)
8                                 ((lambda (G6)
9                                    ((lambda (G7)
10                                      (get (lambda (G4) (+ G3 G4))
11                                           G6 G7))
12                                    c)) data3))
13                               G11 G12))
14                          G14 G15))
15                  b)) data2))
```

```
16                  G19 G20))
17        a))
18  data1)
```

ソースコード 4.5 では以下の順番でプログラムが評価される.

1. data1 の評価結果が G19 に渡される. (1 行目)
2. a の評価結果が G20 に渡される. (2 行目)
3. (get G19 G20) の評価結果が G11 に渡される. (3 行目)
4. data2 の評価結果が G14 に渡される. (4 行目)
5. b の評価結果が G15 に渡される. (5 行目)
6. (get G14 G15) の評価結果が G12 に渡される. (6 行目)
7. (+ G11 G12) の評価結果が G3 に渡される. (7 行目)
8. data3 の評価結果が G6 に渡される. (8 行目)
9. c の評価結果が G7 に渡される. (9 行目)
10. (get G6 G7) の評価結果が G4 に渡される. (10 行目)
11. (+ G3 G4) が評価される. (10 行目)

つまり各変数の値は以下のようになる.

- G19=data1
- G20=a
- G11=(get data1 a)
- G14=data2
- G15=b
- G12=(get data2 b)
- G3=(+ (get data1 a)(get data2 b))
- G6=data3
- G7=c
- G4=(get data3 c)

Scheme の評価規則では関数呼び出し式の場合はまずはじめに引数が評価されるが引数が複数あった場合の評価の順番は特に決められていない. CPS 変換を行う事によりあいまいになっていた評価順序が明確になるという効果もある.

オンライン型の場合は CPS 変換したものと同じものを書く必要がある. 完全オフライン型の場合はコールバック関数を必要としない. 準オフライン型の場合はコールバック関数を必要とする.

4.5 アプリケーション実行までの流れ

本研究においてはアプリケーションを実行するまでに 4 つの段階を踏む必要がある. その流れを以下に示す.

1. プログラマがコールバック関数を伴わない形でプログラムを記述する.
2. プログラマが記述したコードに対してプリフェッチコードの自動挿入を行う.
3. プリフェッチコードを自動挿入したプログラムに対して CPS 変換を行う.
4. JavaScript で記述された Scheme インタプリタを用いて Web ブラウザ上で実行する.

1 においては例えばソースコード 4.2 のようなコードを記述する. 2 においてはソースコード 4.3 のように記述された情報から適切な位置を決定し, その位置にプリフェッチコードを自動的に挿入する. 3 においてはプリフェッチコード挿入後のコードに対して CPS 変換を行う事でコールバック関数相当の処理が後続に渡されるようにする. 4 においては 3 で CPS 変換されたコードを JavaScript で記述された Scheme インタプリタで読み込む事で Web ブラウザ上で実行する. 以上の 4 つの段階を踏む事によりプリフェッチコードおよびコールバック関数が含まれていない Scheme プログラムを, プリフェッチコードとコールバック関数が含まれる JavaScript プログラムとして実行する事ができる.

第 5 章

提案手法の実装

5.1 プリフェッチコードの自動挿入器

本研究においてはプリフェッチコードの自動挿入を行う. そのためのコード挿入器を実装する. 挿入対象は, プログラム中に存在するプログラマによって指定された関数名の関数呼び出し式の直後である.

Scheme プログラムにおいては特殊形式以外は関数呼び出し式であるので, プリフェッチコードの自動挿入を行う際には, リストの先頭部分を見て, 特殊形式か関数呼び出し式かを判断する. もし関数呼び出し式かつ与えられた関数名であれば, その直後にプリフェッチコードを挿入する.

プリフェッチコードの自動挿入器の実装に Scheme の処理系の一つである Gauche[5] を使用した. Gauche は Scheme の標準仕様以外にも様々な便利な機能が追加された実用性のある Scheme 処理系である. プリフェッチコード自動挿入の概略を図 5.1 に示す.

また, プリフェッチコードの自動挿入は 3 つの段階からなる. 以下にそれを示す.

1. 挿入前のコードを 1 行ずつ読み込み, Gauche におけるハッシュテーブルに一つずつ格納する. この際ハッシュテーブルの key として行番号, value として読み込んだテキストとする.
2. ハッシュテーブルを 1 から順番に探索し, value の中にプログラマが指定した関数名をもつ関数呼び出し式があればその直後にプリフェッチコードを挿入するようにハッシュテーブルの中身を書き換える. プリフェッチコードは関数呼び出し式であり, 実引数として取得するデータの URL, および local Storage に格納するときに使用する local Storage のキーを与える.
3. ハッシュテーブルの中身をファイルに書き込む.

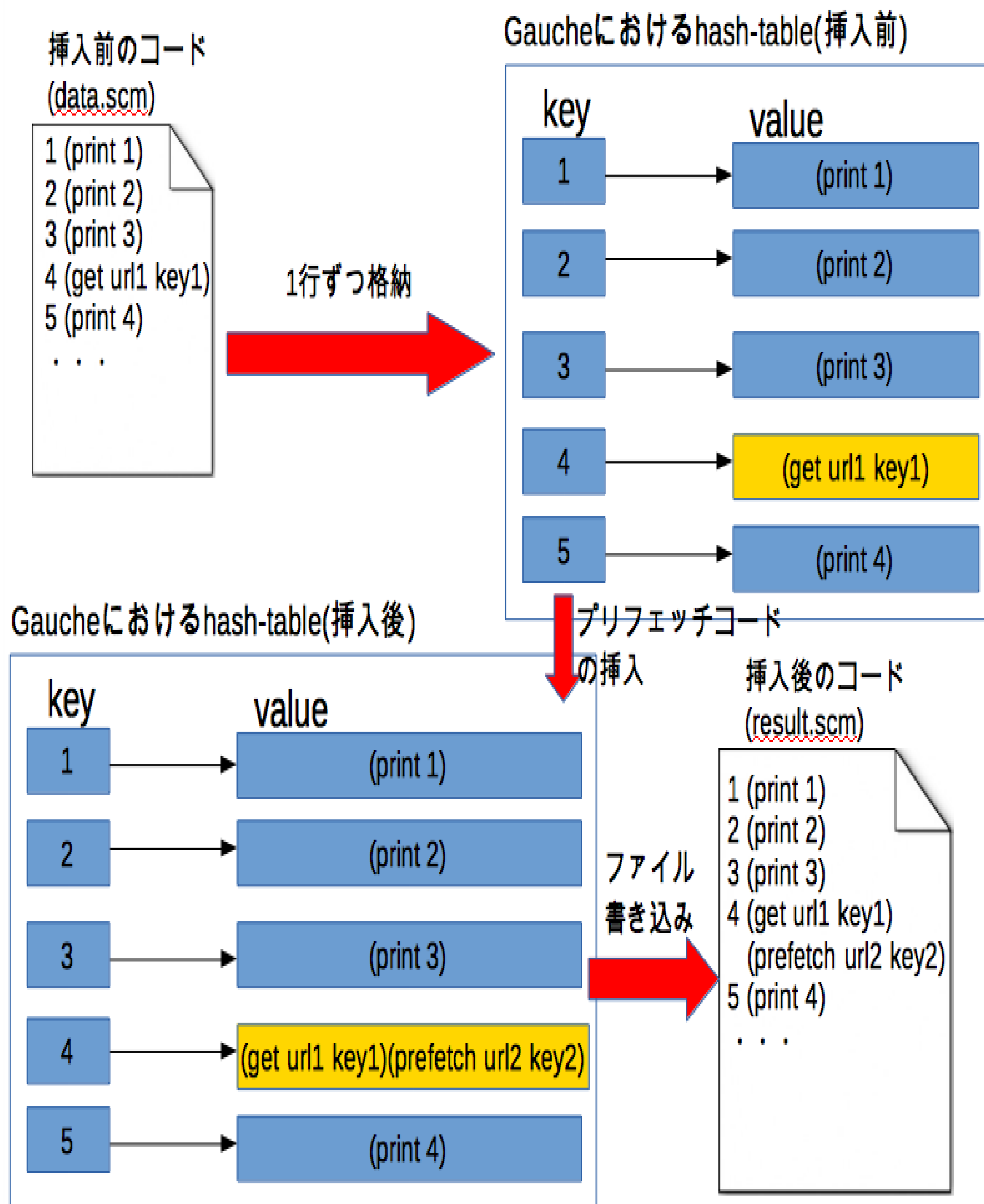

図 5.1　プリフェッチコード自動挿入の概略

しかしながらこの変換器が正しく変換できるプログラムの構造は限られているという問題がある．

5.2 CPS 変換

本研究において CPS 変換を行い, 関数呼び出し後の残りの計算を関数（コールバック関数相当）として, 次の処理を表す関数の第 1 引数に与える必要がある． そのため本研究において CPS 変換を行うプログラムを用意した.

本研究で使用する CPS 変換器は convert という関数の第 1 引数に, CPS 変換を行いたいプログラムを記述する事で CPS 変換を行う事が出来る． 例えばソースコード 5.1 のような形で記述する.

ソースコード 5.1 CPS 変換器の使用例

```
1 (convert '(+ (+ (get data1 a) (get data2 b)) (get data3 c)) stop)
```

ソースコード 5.1 においては CPS 変換を行う対象として (+ (+ (get data1 a) (get data2 b)) (get data3 c)) を指定している． このため (+ (+ (get data1 a) (get data2 b)) (get data3 c)) が CPS 変換される事になる． (+ (+ (get data1 a) (get data2 b)) (get data3 c)) に対して CPS 変換を行った結果は前章のソースコード 4.5 の通りである.

5.3 JavaScript で記述された Scheme インタプリタ

本研究においては CPS 変換の適用の容易さのため Scheme 言語 [6] を対象言語とした． Web ブラウザ上で動作する Scheme 処理系は既存のものを使用し，5.3.2 節で述べる関数をその処理系に加えることで提案手法を実現した.

5.3.1 Scheme インタプリタが解釈できる特殊形式

Scheme インタプリタはいくつかの特殊形式と関数呼び出し式を解釈する事が可能である． Scheme インタプリタが解釈できる特殊形式は以下の通りである.

- (lambda (<パラメータ>...) <式>...)
- (begin <式>..)
- (if <式> <式> [<式>])
- (quote <データ>)
- (let ((<変数名> <式>) ...) <式>...)
- (define (<関数名> <パラメータ> ...) <式>...)
- (time <式>)

特殊形式 lambda は関数を作成する時に使用する．特殊形式 begin は, 与えられた式を順番に評価し, 最後に評価した結果を返す．特殊形式 if は条件式を記述するときに使用する．特殊形式 quote は＜データ＞を式ではなくリテラルデータとして使用したい時に使用する．特殊形式 define は関数定義や変数定義の時に使用する．特殊形式 time は与えられた式を実行した際の実行時間を計る時に使用する．

5.3.2 Scheme インタプリタが解釈できる組み込み関数

オフライン Web アプリケーション開発の為に幾つかの組み込み関数を扱えるようにした．代表的なものを幾つか示す．

(setLS **＜パラメータ＞ ＜パラメータ＞**)

setLS は local Storage にアクセスするための組み込み関数である．setLS は引数として local Storage に値を保存するときに必要となるキーおよび値を指定する．ソースコード 5.2 に実装を示す．

ソースコード 5.2 setLS の実装

```
1 function setLS(key, value) {
2     localStorage.setItem(key, value);
3     return true;
4 }
```

(getLS **＜パラメータ＞**)

getLS は local Storage にアクセスするための組み込み関数である．getLS には引数として local Storage にアクセス時に必要となるキーを指定する．ソースコード 5.3 に実装を示す．

ソースコード 5.3 getLS の実装

```
1 function getLS(key) {
2     return localStorage.getItem(key);
3 }
```

(xhr **＜式＞ ＜パラメータ＞**)

xhr は Web サーバとの通信を行い, データを取得する組み込み関数である．xhr には引数として関数 k(CPS 変換で生成された継続) および web サーバへアクセスするときに必要となる URL を指定する．ソースコード 5.4 に実装を示す．

ソースコード 5.4　xhr の実装

```
function xhr(k, url) {
    var xhrObject = new XMLHttpRequest();
    xhrObject.responseType = "text";
    xhrObject.onreadystatechange = function() {
        if (xhrObject.status === 200 && xhrObject.readyState === 4) {
            var args = [xhrObject.response];
            scmeval_sequence(k.body, k.env.extend(k.args, args));

        }
    };
    xhrObject.open("GET", url, true);
    xhrObject.send(null);
    return true;
}
```

(get **<式> <パラメータ> <パラメータ>**)

get は local Storage からのデータ取得, および Web サーバからのデータ取得を行う組み込み関数である．local Storage 内に必要なデータが存在する時は local Storage に対するアクセスとして働き，もし local Storage 内に必要なデータが存在しなければ Web サーバへのアクセスとして機能する．get は引数としてコールバック関数 k, Web サーバアクセス時に必要となる URL, local Storage アクセス時に必要となるキーを指定する．ソースコード 5.5 に実装を示す.

ソースコード 5.5　get の実装

```
function get(k, url, key) {
    if (navigator.onLine) {
        xhr(k, url);
        return true;
    } else {
        return localStorage.getItem(key);
    }
}
```

(mouseup **<パラメータ> <パラメータ> <パラメータ>**)

mouseup はマウスボタンを (一度押した後に) 離した際のイベントをイベントハンドラとして記述できる組み込み関数である．引数にはイベントハンドラを表す関数 ev, および HTML 上の対象となる要素の id およびイベント発生後の処理を実行するときに必要となる引数 a を記述する．ソースコード 5.6 に実装を示す.

ソースコード 5.6 mouseup の実装

```
function mouseup(ev, id, a) {
    var element = id["name"];
    var div = document.getElementById(element);
    var args = [];
    args.push(a);
    //マウスボタンを離した時に呼び出されるイベント処理.
    div.onmouseup = function() {
        scmeval_sequence(ev.body, ev.env.extend(ev.args, args));
    };
    return true;
}
```

(mousemove **<パラメータ> <パラメータ> <パラメータ>**)

mousemove はマウスを動かした時のイベントをイベントハンドラとして記述できる組み込み関数である．引数にはイベントハンドラを表す関数 ev, および HTML 上の対象となる要素の id およびイベント発生後の処理を実行するときに必要となる引数 a を記述する．ソースコード 5.7 に実装を示す.

ソースコード 5.7 mousemove の実装

```
function mousemove(ev, id, a) {
    var element = id["name"];
    var div = document.getElementById(element);
    var args = [];
    args.push(a);
    //マウス移動時に呼び出されるイベント処理.
    div.onmousemove = function() {
        scmeval_sequence(ev.body, ev.env.extend(ev.args, args));
    };
    return true;
}
```

(mousedown **<パラメータ> <パラメータ> <パラメータ>**)

mousedown はマウスボタンを押した時のイベントをイベントハンドラとして記述できる組み込み関数である．引数にはイベントハンドラを表す関数 ev, および HTML 上の対象となる要素の id およびイベント発生後の処理を実行するときに必要となる引数 a を記述する．ソースコード 5.8 に実装を示す.

ソースコード 5.8 mousedown の実装

```
function mousedown(ev, id, a) {
    var element = id["name"];
```

```
3      var div = document.getElementById(element);
4      var args = a;
5      //マウスボタンを押した時に呼び出されるイベント処理.
6      div.onmousedown = function() {
7          scmeval_sequence(ev.body, ev.env.extend(ev.args, args));
8      };
9      return true;
10 }
```

(mouseclick **<パラメータ> <パラメータ> <パラメータ>**)

mouseclick はマウスをクリックした時のイベントをイベントハンドラとして記述できる組み込み関数である．引数にはイベントハンドラを表す関数 ev, および HTML 上の対象となる要素の id およびイベント発生後の処理を実行するときに必要となる引数 a を記述する．ソースコード 5.9 に実装を示す.

ソースコード 5.9　mouseclick の実装

```
1  function mouseclick(ev, id, a) {
2      var element = id["name"];
3      var div = document.getElementById(element);
4      var args = a;
5      //マウスをクリックした時に呼び出されるイベント処理.
6      div.onclick = function() {
7          scmeval_sequence(ev.body, ev.env.extend(ev.args, args));
8      };
9      return true;
10 }
```

第 6 章

今後の課題

今後の課題としてまず次に必要となるデータを予測して, そのデータを取得できるコードを自動生成する事が挙げられる. 現状では次に必要となるデータの URL を生成し, その URL にアクセスするコードを記述する事は全てプログラマに任せている. 従って本研究の成果ではプログラマが記述したデータを取得するコードを, プリフェッチコードとしてプログラムのロジックの中に挿入する事はできるがそれ以上の事はできない. この問題を解決する方法として, 宣言的にプリフェッチに必要な情報を記述し, その情報をもとにプリフェッチを行うコードを自動生成するという方法が考えられる. 今後この問題を解決する事が必要であると考えている.

また今回は実用的なアプリケーションを実際に作成し評価する事まではできていないので, ある程度の規模の実用性のあるアプリケーションを作成する必要があると考えている. 例えば地図アプリケーションなどがその例として挙げられる.

第 7 章

おわりに

本研究では準オフライン型の Web アプリケーション開発におけるプリフェッチコード挿入に伴うコールバック関数の修正コストの削減手法の提案を行った．また Web サーバとの通信時にコールバック関数を記述する必要のないプログラム記述の手法の提案も合わせて行った．本研究の成果により準オフライン型の Web アプリケーション開発時におけるプログラムの記述コストや修正コストが少しでも軽減できれば幸いである．

謝辞

本研究を遂行するにあたっては, いろいろな方々にお世話になりました.

まず, 指導教員の小宮常康先生には研究の方向性や, プログラムの実装方法, また関連論文の選定に至るまで日頃から大変お世話になりました.

また多田好克先生には基盤ソフトウェア学講座の会合において適切な助言を頂きました.

基盤ソフトウェア学講座の学生諸氏には日頃の大学院での生活や, 研究面において様々なサポート, 助言を頂きました.

最後にこれらの皆さんに感謝いたします.

参考文献

[1] “HTML5,” *http://www.w3.org/TR/html5/*, Jan., 2015.

[2] “Offline first,” *http://blog.joelambert.co.uk/2012/11/26/offline-first-a-better-html5-user-experience/*, Jan., 2015.

[3] “Doloto,” *http://research.microsoft.com/en-us/projects/doloto/*, Jan., 2015.

[4] “Google Gears,” *https://code.google.com/p/gears/*, Jan., 2015.

[5] “Gauche,” *http://practical-scheme.net/gauche/index-j.html*, Jan., 2015.

[6] “The Revised^6 Report on the Algorithmic Language Scheme,” *http://www.r6rs.org/*, Jan., 2015.

[7] Yung-Wei Kao, ChiaFeng Lin, Kuei-An Yang and Shyan-Ming Yuan: “A Web-based, Offline-able, and Personalized Runtime Environment for executing applications on mobile devices,” *Computer Standards & Interfaces*, Vol.34, No.1, pp.212-224, Jan., 2012.

[8] Weinsberg, Y. and Ben-Shaul, I.: “A programming model and system support for disconnected-aware applications on resource-constrained devices,” *Proceedings of the 24th International Conference on Software Engineering (ICSE ’02)*, pp.374-384, May., 2002.

[9] Benjamin Livshits and Emre Kcman: “Doloto: Code Splitting for Network-Bound Web 2.0 Applications,” *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (SIGSOFT ’08/FSE-16)*, pp.350-360, Nov., 2008.

[10] Marc Feeley.: “Polling efficiently on stock hardware,” *Proceedings of the conference on Functional programming languages and computer architecture (FPCA ’93)*, pp.179-187, 1993.

[11] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams: “ORBIT: an optimizing compiler for scheme,” *Proceedings of the 1986 SIGPLAN symposium on Compiler construction (SIGPLAN ’86)*, pp.219-233, 1986

[12] Guy L. Steele, Jr.: “Rabbit: A Compiler for Scheme,” *AI Memo474, MIT*, 1978

[13] 田中義純, 田浦健次朗, 米澤明憲: “定期的なポーリングを保証するアルゴリズム”, 並列処理シンポジウム JSPP2001, Jun., 2001