情報処理学会論文誌 Vol.54 No.2 647–658 (Feb. 2013)

動的Webアプリケーションのためのプリフェッチ機構

柴田 和祈[1,†1,a)] 高田 眞吾[1,b)]

受付日 2012年5月9日, 採録日 2012年11月2日

概要：Web アプリケーションの高速化の一技術としてプリフェッチがある．プリフェッチとは，Web ページをサーバ側からクライアント側へと先取りすることによって，ページ遷移にかかる時間を短縮する機能である．プリフェッチには様々な手法がある．従来の手法は Web ページの遷移が固定である静的な Web アプリケーションにのみ対応しており，ユーザの入力によって遷移先の Web ページが変化する動的な Web アプリケーションには対応していない．しかし，現在，静的なページのみで構成されている Web アプリケーションはごく少数であり，動的な Web アプリケーションのためのプリフェッチ手法はない．その最大の原因は，ユーザのクリック先は予測できても，ユーザがフォームなどにおいて入力する内容まで予測することができないからである．本研究では動的な Web アプリケーション（PHP アプリケーション）のためのプリフェッチ機構を提案する．提案機構の基本コンセプトは，Web ページを静的なコンテンツと動的なコンテンツに分離することである．分離後，静的コンテンツはリンクプリフェッチを用いてあらかじめ取得し，動的コンテンツは Ajax を用いて後から補完することで，動的な Web アプリケーションにおけるプリフェッチを可能にする．

キーワード：動的 Web アプリケーション，プリフェッチ

A Prefetching Framework for Dynamic Web Applications

Kazuki Shibata[1,†1,a)] Shingo Takada[1,b)]

Received: May 9, 2012, Accepted: November 2, 2012

Abstract: Prefetch is a technique to “speed-up” Web applications. It can save the time needed for page transition by downloading in advance Web pages from the server to the client. There are various approaches for prefetching. The conventional approach can handle static Web applications, but not dynamic ones. However, few applications are composed solely of static pages. Thus, we need to be able to prefetch dynamic Web applications. This is difficult because although it may be possible to predict where a user clicks, it is not possible to predict what a user will input into a form, etc. We propose an approach for prefetching dynamic Web applications (PHP applications). The basic idea is to divide Web pages into static and dynamic contents. The static contents can be obtained in advance by using link prefetch, and the dynamic ones can be complemented later by using Ajax.

Keywords: dynamic Web application, prefetch

1. はじめに

Web は多くの人に利用されており，ネットワークのトラフィックはしきりに増加している．技術の進歩により通信の帯域幅は増加したが，それでも混み合った回線によってはページのロードに時間がかかる [1]．

Web アプリケーションのレスポンスタイムは非常に重要である．まず，待ち時間は収益に影響を与える．Linden [2] によると，Google では検索結果を返す時間が 0.5 秒遅延するだけでトラフィックや収益が 20%も落ちてしまい，Amazon が実施した 100 ミリ秒単位でページを遅延させる実験において，非常に小さな遅延ですら収入に大きく響く．

[1] 慶應義塾大学
Keio University, Yokohama, Kanagawa 223–8522, Japan
[†1] 現在，ヤフー株式会社
Presently with Yahoo Japan Corp.
[a)] shibata@doi.ics.keio.ac.jp
[b)] michigan@ics.keio.ac.jp

さらに，Web アプリケーションに限らず，コンピュータシステムのレスポンスタイムはユーザビリティに影響を与える．Nielsen [3] によると，0.1 秒はシステムが瞬間的に反応しているとユーザが感じる限界の時間である．1.0 秒はユーザの思考の流れが途切れない限界の時間であるが，データを直に操作しているという感覚は失われる．10 秒はユーザが画面との掛け合いに集中できる限界の時間であり，これを超えると，ユーザはコンピュータの処理が終わるまでの間，他のタスクをしようとする．

このように，Web アプリケーションの処理の結果が Web ページに表示されるまでの時間を短縮することには大きな価値がある．この時間は大きく以下の順番で発生する．

(1) クライアントからサーバへリクエストを送信する通信時間
(2) リクエストを受け取ったサーバ上の Web アプリケーションにおける処理時間
(3) サーバからクライアントへレスポンスを送信する通信時間
(4) クライアント上でレスポンス（Web ページ）を表示するための時間

プリフェッチと呼ばれる高速化技術では，(3) のレスポンスを事前に送る—つまり，Web ページをサーバ側からクライアント側へと先取りする—ことによって，ページ遷移にかかる時間を短縮する機能を指す [4]．

プリフェッチについては，ユーザからの入力によりレスポンスが変わらないような静的な Web アプリケーションについては一定の効果があがっている [1], [4], [5]．しかし，現在静的なページのみで構成されている Web アプリケーションはごく少数であり，ほとんどの Web アプリケーションは動的である．ここでいう動的な Web アプリケーションとは，ユーザの入力により表示される Web ページが変わるようなアプリケーションのことである．しかし，ユーザの入力を予測することは困難であり，プリフェッチを行うことができない．そこで，本研究では動的な Web アプリケーションにおけるプリフェッチ機構を提案する．特に，本研究では動的な Web アプリケーションのうち，PHP で記述された Web アプリケーションを対象とする．

以下，2 章では，プリフェッチに関する関連研究を示す．3 章において，動的 Web アプリケーションを対象にしたプリフェッチ機構の提案を行う．4 章では評価実験を示し，考察する．最後に，5 章でまとめる．

2. 関連研究

本章では，プリフェッチの関連研究をプリフェッチ対象の決定と，プリフェッチ対象の指示の方法に分けて示す．

2.1 プリフェッチ対象の決定

プリフェッチ対象の決定方法に関する研究として，クライアント側で決定する Dahlan らの研究 [5] と，サーバ側で決定する de la Ossa らの研究 [1] がある．

Dahlan らは，クライアント側でプリフェッチ対象を決める手法 Asynchronous Predictive Fetch（APF）を提案した [5]．APF は，ページ遷移の起こらないユーザアクション（マウスオーバやチェックボックスのチェックなど）に着目し，ユーザの現在のアクションから次のアクションを予測する．たとえば，マウスオーバやドロップダウンリストのチェックを検知した際に，それをプリフェッチ対象とする．しかし，たまたまマウスオーバをしてしまうということもありうるため，このままでは正確さに欠ける．そこで，マウスオーバをしてから数ミリ秒の時間差を設けてからプリフェッチを行う．また，プリフェッチ中に他のリンクをマウスオーバした場合，そのリンク先もプリフェッチ対象とし，新たにプリフェッチを行う．この際，実行中のプリフェッチは破棄されないため，余計な帯域幅が必要となる．

de la Ossa らは，サーバ側において，ユーザのアクセス履歴を分析することで動的にプリフェッチ対象を決定し，クライアント側に通知する手法を提案した．提案手法の最大の特徴はプリフェッチ対象を決めるタイミングである．通常は，ユーザが実際にアクセスした Web ページに基づいてプリフェッチ対象を決定するのに対して，de la Ossa らの手法ではプリフェッチで取得した Web ページに基づいてプリフェッチ対象を決めている．つまり，ユーザが実際にアクセスする前にプリフェッチ済みの Web ページから，次にプリフェッチすべき Web ページを決め，実際にその Web ページを取得する．

2.2 プリフェッチ対象の指示

Fisher らは，リンクプリフェッチという，Web ページの先読み対象をブラウザに通知する手法を提案した [6]．ブラウザに通知する手段として，(1) HTML の <link> タグによる指定，(2) HTML の <meta> タグによる指定，(3) HTTP レスポンスのリンクヘッダによる指定の 3 つがある．たとえば，big.jpg というファイルをプリフェッチしたい場合，それぞれ次のように行う．

- <link> タグによる指定：
  <link rel=“prefetch” href=“big.jpg”>
- <meta> タグによる指定：
  <meta HTTP-EQUIV=“Link”
  CONTENT=“<big.jpg>; rel=prefetch”>
- リンクヘッダによる指定：
  Link: <big.jpg>; “rel=prefetch”

ブラウザはこれらの先読みリクエストを待ち行列に入れ，順にサーバから取得する．また，ヘッダによる指定の方がプリフェッチの中止はしやすく，プリフェッチに対応していないブラウザであっても問題はない．

2.3 既存のプリフェッチ手法の問題

Dahlan らの手法および de la Ossa らの手法は，いずれも静的 Web アプリケーションを対象にしているため，たとえばフォームの入力により遷移先ページが変わるような動的 Web アプリケーションは扱えない．Dahlan らの手法は，さらにクライアント側で実装するために，ブラウザに専用ソフトウェアを組み込まなくてはならない．そのため，ユーザの入力により内容が変わるような Web ページに対して対応する必要がある．

Fisher らの手法も特に動的 Web アプリケーションに直接対応しているわけではない．また，Web ブラウザが通知された情報を解釈し，それに基づいてサーバからプリフェッチ対象を取得できなければならないが，現在 Firefox などの Web ブラウザはリンクプリフェッチを標準的にサポートしている．

3. 提案：動的 Web アプリケーションのプリフェッチ機構

本章では，動的な Web アプリケーションのためのプリフェッチ機構を提案する．本機構の主要部分はサーバ側にあり，クライアント側に対して特別なアッドオンをしなくても利用できる．なお，提案機構は PHP アプリケーション（Web アプリケーション中の PHP ファイル）を対象とする．

3.1 概要

ユーザのクリック先は予測できても，ユーザが直接入力する内容までは予測することはできない．そこで，本提案機構は次の 2 点を基本方針とする．

- 静的コンテンツはリンクプリフェッチを用いて先に取得する．
- 動的コンテンツは Ajax（Asynchronous JavaScript & XML）の技術を用いて後から補完する．

サーバから Web ブラウザへのプリフェッチ対象の指定，および，指定されたプリフェッチ対象の取得については，既存技術を利用する．具体的には，Fisher らが提案したリンクプリフェッチ [6] を利用し，Web ブラウザが Firefox などリンクプリフェッチに対応していることを前提とする．また，どのファイルをプリフェッチ対象にするかを指定するために，.htaccess ファイルを利用するため，サーバは Apache であることを前提とする．

提案機構の処理（**図 1** と**図 2**）は，PHP アプリケーションに対する事前処理，プリフェッチ対象に関する情報の定期的な更新処理，アプリケーション実行中に行われる処理の大きく 3 種類がある．

PHP アプリケーションに対する事前処理が本提案の中心である．事前処理（図 1）として，PHP アプリケーションを静的コンテンツと動的コンテンツに分離する処理（3.2 節）と，データ渡しに関する JavaScript コードの挿入（3.3 節）がある．後者のデータ渡しは，Cookie を用いて，実行中にブラウザから入力した情報を，キャッシュにある静的コンテンツにデータを渡したり，サーバ側にあるアプリケーションに渡したりするためであり，図 2 ③ ④ ⑤ を可能にする．事前処理が終わった後，両コンテンツを Web サーバ上にデプロイする．

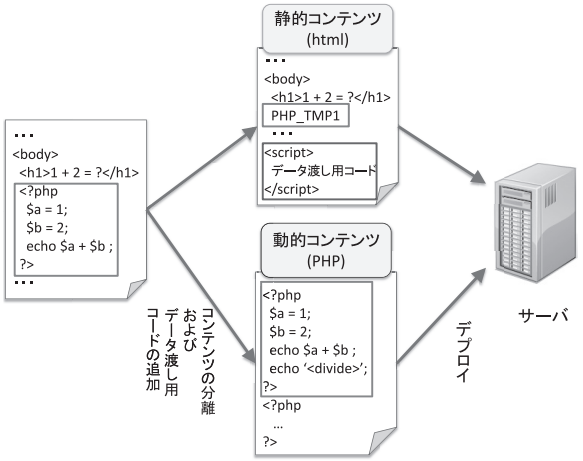


図 1 提案機構における事前処理
Fig. 1 Preprocessing.

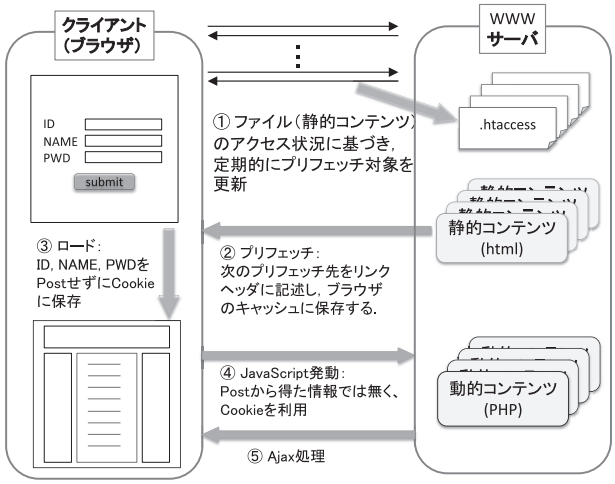


図 2 提案機構における実行時処理
Fig. 2 Runtime processing.

各静的コンテンツに対するプリフェッチ対象は，過去のアクセス履歴を参考に決定する．この情報は，.htaccess ファイルに記載し，アクセスの仕方の変化に対応するため定期的に更新する（図 2 ①；3.4.2 項）．

アプリケーション実行中の処理については，静的コンテンツのプリフェッチと動的コンテンツの挿入がある．静的コンテンツのプリフェッチについては，従来のリンクプリフェッチの手法と同様に，ある静的コンテンツがプリフェッチ対象と指定されていれば，クライアントのブラウザにその静的コンテンツを送信し，ブラウザのキャッシュに保存する（図 2 ②；3.4 節）．

動的コンテンツの挿入については，事前処理の時点で静的コンテンツに挿入した JavaScript コードを実行することにより実現する．たとえば，フォームに入力がなされ，submit ボタンが押されると，ブラウザのキャッシュに保存した静的コンテンツをロードする（図 2 ③）．その際，フォームに入力された情報は Cookie に保存する．次にその静的コンテンツ内に埋め込まれている JavaScript コードが発動し（図 2 ④），Cookie に保存した情報とともに Web サーバをアクセスし，Ajax 処理によって動的コンテンツを取得する（図 2 ⑤）．このようにして，Ajax という既存技術を利用して，動的処理部分を後からページに挿入する．

本章の残りでは，まず事前処理で行った静的コンテンツと動的コンテンツの分離について述べる．次に，ユーザが入力した情報をサーバに送り結果を表示する処理のために静的コンテンツに挿入される JavaScript コードについて述べる．最後にプリフェッチについて述べる．プリフェッチ手法そのものはリンクプリフェッチを利用するが，プリフェッチ対象を決めるために行うページ遷移の分析，および，プリフェッチ対象を保存するために利用する .htaccess ファイルについて述べる．

3.2 静的・動的コンテンツの分離

静的コンテンツおよび動的コンテンツは別々に取得するため，事前処理として，Web ページを静的なコンテンツと動的なコンテンツにまず分離する必要がある（図 1）．

静的・動的コンテンツの分離は次のように行われる．PHP アプリケーションにおいて，動的な処理は “<?php” と “?>” に囲まれた部分で行われることに着目し，まず静的処理部分と動的処理部分を分離し，それぞれ独立のファイルとして保存する．その際，静的ファイルの方には，元々動的処理が記述されていた部分に “PHP_TMP 番号” という文字列を挿入する．

次に静的ファイルには後に Ajax 処理を行うための JavaScript コードを挿入し（3.3 節），動的ファイルには “<?php” と “?>” に囲まれた部分ごとの区別を付けるためにそれぞれの語尾に <divide> というタグを挿入する．

なお，元々のアプリケーションは PHP ファイルどうしがリンクされているため，静的・動的コンテンツを分離する過程で，リンク関係が不完全となる．そこで新しくリンクをつなぎ直す必要がある．Web アプリケーションを実行する際，最初に読み込まれるのは静的ファイルの方であるため，リンク関係の再構築は静的ファイルを中心に行う必要がある．具体的には，<a> タグの href 属性と <form> タグの action 属性に注目し，たとえば，元々 b.php へのリンクに対しては，b.html へのリンクにする（**図 3**）．

3.3 データの受け渡しのための JavaScript コード挿入

本研究では特に動的 Web アプリケーションを対象にし

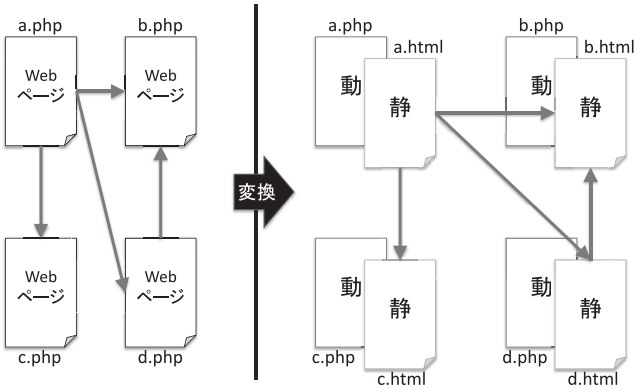


図 3 リンク関係の再構築
Fig. 3 Reconnecting the links.

```
<form method="POST" action="a.php">
  <input type="text" name="id" />
  <input type="submit" value="送信" />
  <form method="POST" action="b.php">
    <input type="text" name="content" />
    <input type="submit" value="送信" />
  </form>
</form>
```

図 4 ボタンによるページ遷移
Fig. 4 Page transition via button.

ているため，ユーザ入力を扱える必要がある．しかし，静的コンテンツがキャッシュにあったり，動的コンテンツがサーバ側にあったりするため，ユーザ入力によるデータの受け渡しの仕組みが必要となる．本提案では，データの受け渡しの仲介役として Cookie を利用し，次の処理を JavaScript コードとして事前に静的コンテンツに挿入する．

(1) ユーザ入力を Cookie に保存する（図 2 ③ の処理）．
(2) ページ遷移後，Cookie を読み込む．
(3) 読み込んだ Cookie の内容を，JavaScript の非同期通信を利用してサーバに送信する（図 2 ④ の処理）．
(4) Ajax により結果を取得したら，表示する（図 2 ⑤ の処理）．

次にボタンクリックを例に，挿入される JavaScript コードを一部具体的に示す．なお，JavaScript コードはいずれも jQuery [7] という JavaScript ライブラリを用いている．また，キャッシュに該当する静的コンテンツが保存されていない場合は，通常の Web ページアクセスを行う．

3.3.1 ユーザ入力の Cookie への保存処理

ページ遷移の種類によってデータの受け渡し方法が変わる．たとえば，一般的に，ボタンクリックを通してページ遷移をする場合，POST を用いる．**図 4** では，id が a.php に，content が b.php に POST で渡される．

HTML のボタン要素は 2 種類の方法で作成可能である．1 つ目は <button> タグを用いる方法で，もう 1 つは <input> タグの type 属性に button，submit，または reset を与え

```
1: $(":submit").click(function() {
2:     var elements = $(":input");
3:     for (var i = 0; i < elements.length; i++) {
4:         if(elements[i].getAttribute("type") != "submit"){
5:             document.cookie = elements[i].getAttribute("name")
                                    + "=" + elements[i].value + ";";
6:         }
7:     }
8: });
```

図 5 ユーザ入力を cookie に保存する処理に関する JavaScript コード
Fig. 5 JavaScript code for saving user input to Cookie.

た場合である．これらの要素について，DOM（Document Object Model）からアクセスし，onClick イベントハンドラを設定する．これにより，ボタンがクリックされた際に通常の処理を中断し，サブルーチンに処理を投げることができる．ボタンクリックによってページ遷移が起こる際の，データの受け渡しの流れは次のようになる．

(1) onClick イベントが発生した際に，Form 部品（input，textarea など）の name 属性と value 属性を探索する．
(2) DOM を用いて，本来サーバに送るべきデータを取得する．
(3) 取得したデータを cookie に保存する（有効期限はページ遷移直後）．
(4) ページを遷移する．つまり，キャッシュに保存されている静的コンテンツを Web ブラウザ上に表示する．

上記の手順 (1)〜(4) までは図 2 の③に該当する．これらの処理を行うために，静的コンテンツに挿入される JavaScript コードを**図 5** に示す．

jQuery のセレクタ（1 行目の “:submit” および 2 行目の “:input”）によって，HTML コード内の特定の要素にアクセスが可能となる．submit セレクタと click 関数によって，submit ボタンがクリックされた際に 1 行目の無名関数が呼ばれる．2 行目では，jQuery の input セレクタを用いて，すべてのフォーム部品（input，text area，select，button）を選択する．4–5 行目で，フォーム部品のうち type 属性が submit 以外の要素であった場合は Cookie に書き込みを行う．

3.3.2 ページ遷移後の Cookie の読み込み処理

キャッシュにあった WWW ページ（静的コンテンツ）への遷移が終わったら，前項で Cookie に保存したユーザ入力を読み込み，サーバへ送る準備を行う．読み込みの JavaScript コードを**図 6** に示す．

Cookie は key と value によるハッシュ構造になっているので，それぞれを格納するための配列を 1–2 行目で定義する．document.cookie に，すべての Cookie が連なって格納されている．3 行目では “;” で区切ることで，連なりを解除し，一組ごとに分ける．また，日本語などの 2 バイト文字はエンコードして Cookie に保存するので，ここでは “unescape(document.cookie)” によってデコードしている．5 行目で，組ごとになっている Cookie を key と value に分ける作業を行う．6–7 行目で，key と value を用意しておいた配列に格納する．

```
1 : var key = new Array();
2 : var value = new Array();
3 : var cookie_set = unescape(document.cookie).split(';');
4 : for(var i = 0; i < cookie_set.length; i++){
5 :   var set = cookie_set[i].split('=');
6 :   key[i] = set[0];
7 :   value[i] = set[1];
8 : }
```

図 6 ページ遷移後の Cookie の読み込み処理に関する JavaScript コード
Fig. 6 JavaScript code for reading Cookie.

3.3.3 Cookie の内容をサーバに送信し，表示するための処理

Ajax を利用した動的処理情報の取得については以下のとおりである．

(1) 読み込んだ cookie の値を，非同期通信を利用してサーバに送信する．
(2) Ajax により結果が返ってきたら，<divide> という目印を参考に，静的コンテンツ中の各 PHP_TMP を，対応する動的処理結果の文字列で置換する．

上記の手順 (1) は図 2 の④に該当し，手順 (2) は図 2 の⑤に該当する．これらの処理を行うために，静的コンテンツ中に挿入する JavaScript コードを**図 7** に示す．

1 行目では，jQuery の関数を用いて，サーバ側にある *filename* というファイルに対して key と value のセットを送信する．それに対するコールバック関数が 2–7 行目となる．引数として与えた data に，サーバ側で処理した結果が String 型で格納されている．2 行目で，まずサーバ側で処理した結果を <divide> タグで分割する．3 行目で，現在画面に表示されている Web ページ内すべてを取得する．4–6 行目で，ページ内に埋め込まれている “PHP_TMP 番号” を，サーバ側で処理した結果に置換していく．7 行目で，ブラウザに描画を行う，つまり，静的ページに動的処

```
1 : $.post('filename', {key: value},function(data){
2 :   var php_str = data.split('<divide>');
3 :   var str = document.getElementsByTagName('html')[0].innerHTML;
4 :   for(var i = 0; i < php_str.length; i++){
5 :     str = str.replace('PHP_TMP'+i, php_str[i]);
6 :   }
7 :   document.getElementsByTagName('html')[0].innerHTML = str;
8 : });
```

図 7 Cookie の内容をサーバに送信し，結果を表示する処理に関する JavaScript コード
Fig. 7 JavaScript code for sending Cookie information and showing result.

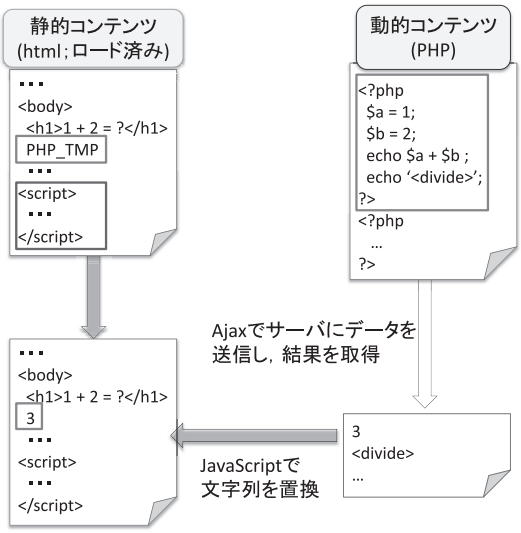


図 8 最終結果の表示
Fig. 8 Final result.

理結果を挿入した最終的な Web ページを表示する．

たとえば，**図 8** では，最初の PHP_TMP の出現を「3」で置き換える．以後，同様に PHP_TMP の出現では，<divide> の後の処理結果の文字列で置き換える．以上より，本来得る予定であったページを，静的コンテンツと動的コンテンツを別ファイルに分けた形で表示可能となる．

3.4 プリフェッチの手順

リンクプリフェッチという機能を利用することで，Web ページの先読み対象をブラウザに通知することができる．リンクプリフェッチを用いたプリフェッチに関する手順は，プリフェッチ対象の指定と，実行時処理に分かれる．

プリフェッチ対象の指定は次の 2 つのステップからなり，定期的に行われる．

(1) Web サーバのログを解析し，ユーザがどのページからどのページへ遷移しやすいのかを分析する．
(2) 分析結果に基づき各ページごとにプリフェッチ対象を決定し，.htaccess ファイルに記述する．

実行時処理に関しては次の 2 つのステップからなる．

(1) サーバは，各リクエストに対して，.htaccess ファイル内の指定に従い，レスポンスにプリフェッチ対象を記述したリンクヘッダを付け加える．
(2) ブラウザはレスポンスを受け取ると，リンクヘッダで指定したファイルをプリフェッチする．

この処理は，.htaccess ファイルからプリフェッチ対象を取得する点以外は Fisher らによるリンクプリフェッチと変わらない．

以下，ページ遷移の分析および .htaccess ファイルの利用について述べる．なお，以下では，アクセスされたファイル名のことを「リクエスト」と呼び，リンク元のページの URL のことを「リファラ」と呼ぶ．

3.4.1 ページ遷移の分析

本提案におけるプリフェッチは，リファラごとに遷移回数の多いリクエストを対象とする．そのために，提案機構では，Web サーバの動作を記録したアクセスログを解析することにより，ユーザのページ遷移状況を把握する．アクセスログの解析において，リファラとリクエストの情報を取得し，リファラとリクエストの表を構築する．

分析手順は大きく，リファラの一覧取得，リクエストの一覧取得，集計処理の 3 つに分かれる．

リファラの一覧はアクセスログファイルを調べ，リファラ用の配列に保存することにより行う．具体的には，アクセスログファイルの各行に対して，まず対象アプリケーションと同じドメインであるリファラを探す．そして，そのリファラがすでにリファラ用の配列に含まれていれば，何もしない．そのリファラがリファラ用の配列に含まれていなければ，配列に追加し，リファラ ID を割り振る．結果として，リファラ用の配列中にリファラの一覧が含まれる．なお，本提案機構では，同一ドメイン内におけるページ遷移のみを対象とするため，リファラは同じドメインのものに限る．

次に，リクエストの一覧の取得は同様の手順で行う．つまり，アクセスログファイルの各行に対して，まず，対象アプリケーションと同じドメインであるリクエストを探す．そして，そのリクエストがすでにリクエスト用の配列に含まれていれば，何もしない．そのリクエストがリクエスト用の配列に含まれていなければ，配列に追加し，リクエスト ID を割り振る．結果として，リクエスト用の配列中にリファラの一覧が含まれる．

表 1 リファラ・リクエスト表の例
Table 1 Example of referrer-request table.

	リクエスト					
リファラ	action.js	a.html	flower.jpg	b.html	d.html	c.html
a.html	23	1	9	17	7	6
b.html	12	17	11	2	9	8
c.html	5	10	8	13	4	2
d.html	35	14	21	6	2	11

最後に，集計処理では，アクセスログファイル，リファラ用配列，リクエスト用配列を調べ，集計用の int 型 2 次元配列 Rank に結果を保存する．具体的には，アクセスログファイルの各行に対して，リファラ用配列からその行に現れるリファラのリファラ ID を取得し，リクエスト用配列からその行に現れるリクエストのリクエスト ID を取得する．そして，Rank に対して，リファラ ID とリクエスト ID を配列の添字として利用し，該当セルに対してインクリメントする．結果として，Rank 中の各値は，該当するリファラとリクエストが何回呼び出しの関係があったかを示すこととなる．たとえば，**表 1** では，a.html から action.js に遷移した回数は 23 回，再度自分に遷移したのは 1 回，flower.jpg は 9 回などとなる．

3.4.2 .htaccess ファイルの利用

ブラウザのリンクプリフェッチ機能を利用して，対象ファイルのプリフェッチを行う．リンクプリフェッチに関しては，2 章で述べたとおり，プリフェッチ対象を指定する方法は複数あるが，本提案機構では HTTP レスポンスのリンクヘッダによる指定を行う．よって，プリフェッチ先を動的に変化させるためには，リンクヘッダを書き換える必要がある．リンクヘッダの書き換えは .htaccess ファイルを利用する．

.htaccess ファイルとは，Web サーバの動作をディレクトリ単位で制御するためのファイルである．具体的には，CGI や SSI などを実行するための宣言（命令）や，拡張子ごとにファイルタイプを指定する MIME タイプの設定，ユーザ認証，IP アドレスやドメイン単位でのアクセス制限などを書き込むことができる．.htaccess ファイルで設定した内容は，.htaccess ファイルがあるディレクトリとそのサブディレクトリに効果があり，効果があるディレクトリに入っているファイルすべてに影響をおよぼす．また，サブディレクトリにも .htaccess ファイルを別に置くことができ，この場合は両方のファイルの効果が発生する．上位ディレクトリの設定と矛盾する際はサブディレクトリの設定を優先する．本提案機構ではアプリケーションの root ディレクトリに .htaccess ファイルを設置する．.htaccess ファイルにおいて，ファイルごとにディレクティブを設けることで，リクエストされるファイルに応じて，レスポンスのリンクヘッダを書き換えることができる．

ユーザのアクセス状況は時間とともに変化していくため，一定時間ごとに .htaccess を書き換える．処理手順は次の 2 つのステップからなる．

(1) 2 次元配列 Rank をリファラごとに調べ，値の高い上位 i 個のリクエストを取得する．
(2) 取得したリファラ，リクエストの組を，.htaccess ファイルに書き込む．

上記手順の i（つまり，プリフェッチするコンテンツ数）についてはアプリケーション管理者が事前に設定する．たとえば，表 1 において，上位 3 つ（$i = 3$）を対象とした場合，a.html に関しては，action.js，flower.jpg，b.html がプリフェッチ対象となり，次が .htaccess に書き込まれる．

```
<Files "a.html">
Header set Link "<action.js>; rel='prefetch'"
Header set Link "<flower.jpg>; rel='prefetch'"
Header set Link "<b.html>; rel='prefetch'"
</Files>
```

上記の記述の場合，サーバ（Apache）は，a.html というファイルのリクエストが来た際，action.js，flower.jpg，b.html の 3 つのファイルをプリフェッチするという情報をレスポンスのリンクヘッダに追加する．追加そのものは Apache サーバの機能をそのまま利用する．そして，ブラウザ側が Firefox などリンクプリフェッチに対応していれば，対象ファイルがプリフェッチされることとなる．

以上により，一定時間ごとにプリフェッチ対象をユーザアクセスに応じて切り替えることができる．

3.5 実装

本提案機構のために，新たに実装したモジュールは，Dynamic Contents Divider（DCD）と mod_predict の 2 つである．

DCD は事前処理において用いるモジュールであり，静的コンテンツ・動的コンテンツの分離（3.2 節）と，データ渡しのための JavaScript コード挿入（3.3 節）を行っている．挿入する JavaScript コードについては，jQuery に基づいた jquery.min.js ファイルを用意し，その中には図 5，6，7 にあるコードなどが含まれている．

これに対して，mod_predict は，Apache のモジュールとして実装し，ページ遷移の分析や．htaccess ファイルの書

き換えを行う（3.4 節）．なお，ページ遷移の分析は Apache のモジュールであるため，アクセスログ中に提案機構によるアクセス情報はない．

最後に，.htaccess ファイルの書き換えなどの処理を定期的に行うために，Cron[*1]を利用する．

4. 評価

本章では，提案機構の評価実験と利用制約について述べる．

4.1 評価実験

4.1.1 評価方法

評価実験では，2 つのアプリケーションを対象に，提案機構を利用した場合と利用しなかった場合におけるロード時間の違いを計測した．

対象アプリケーションは，グループ管理システムとショッピングシステム [8] であった．グループ管理システムはシングルページ（group_kanri.php）からなるアプリケーションであり，903 行のコードからなる．ショッピングシステムはマルチページからなるアプリケーションであり，2,542 行のコードからなる．ショッピングシステムは，4 つの PHP ファイル，16 の画像ファイル，3 つの記録ファイル（ユーザ名とパスワードなど）の 3 種類の合計 23 ファイルから構成される．4 つの PHP ファイル中，実験で使用したのは goods.php である．

ロード時間として次の 2 種類を計測した．

- 元々の（変換前の）アプリケーションに対するページのロード時間
- 提案機構を用いて変換を行ったアプリケーションに対するページのロード時間

なお，ここでいうロード時間とは，「ユーザがボタンまたはリンクをクリックした瞬間から，画面の描画がすべて終わるまで」を指す．これは 1 章で述べた 4 種類の時間の合計となる．本機構における処理の関係上，オーバラップする時間が発生するため，このロード時間で評価を行った．

提案機構を利用したアプリケーションの場合，次のステップを 10 回実行し，測定値の平均をとる．

(1) 対象アプリケーションの事前処理を行い，サーバにデプロイする．
(2) php 以外のすべてのファイル（つまり，静的コンテンツ）をキャッシュに入れる．
(3) 該当 Web ページをブラウザで表示し，該当ボタンをクリックする．
(4) 結果を得たとき，Firebug によるロード時間を計測する．

[*1] UNIX 系システムの常駐プログラムの一種であり，ユーザの設定したスケジュールに基づいて，指定したコマンドやシェルスクリプトなどを自動実行することが可能である．

Firebug [9] は Mozilla Firefox のアドオンであり，Web アプリケーションに関わる様々な情報を取得することができる．事前に静的コンテンツをキャッシュに入れており，また連続的に Web アプリケーションをアクセスしているわけではないため，リファーラ・リクエスト表を用いなかった．

元々のアプリケーションの場合，上記のステップ (1), (2) を実施せずに，サーバにアプリケーションをデプロイする．

評価環境については次のとおりである．

- Web サーバ
  - サーバ PC：Debian 5.0.7
  - Web サーバ：Apache 2.2.9
- クライアントが対象アプリケーションと同一ネットワーク内にある場合（以下，環境 1 と表記）
  - クライアント PC：Windows Vista, Intel Core 2 Duo, 2.00 GB メモリ
  - ネットワーク接続：100 Mbps
- クライアントが対象アプリケーションと同一ネットワーク内にない場合（以下，環境 2 と表記）
  - クライアント PC：Windows 7，AMD Athlon™ II X2 240 Processor，4.00 GB メモリ
  - ネットワーク接続：2.2 Mbps

なお，クライアントと対象アプリケーションが同一ネットワークにないケースは，一般のインターネット越しで行ったため，ネットワーク接続の値は，http://www.sokudo.jp/ を用いて計測した値を用いている．

4.1.2 事前処理の結果

本提案機構では，元の PHP ファイルを html ファイル（静的コンテンツ）と php ファイル（動的コンテンツ）に分離する．分離した結果の容量を**表 2** に示す．表から分かるとおり，グループ管理システムは，ほとんどが動的コンテンツであった．

4.1.3 測定結果と考察

測定結果を**表 3** に示す．なお，静的コンテンツの総容量とは，html ファイルだけではなく，画像ファイルも含む．

表 3 より，グループ管理システムの環境 1 以外ではロード時間の短縮が見られる．グループ管理システムにおいて環境 1，2 を比較すると，同じアプリケーションでもネットワークや PC 環境によって，ロード時間に差が出ることが分かる．

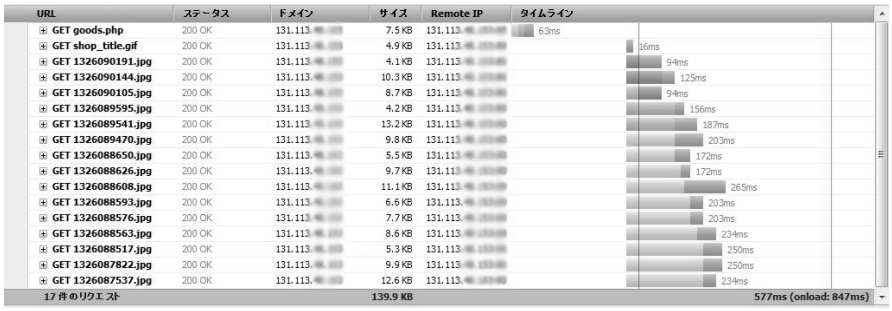
グループ管理システムとショッピングシステムを比較した場合，ショッピングシステムの方がグループ管理システムよりも良い結果が得られた．これは両アプリケーションにおける静的コンテンツの量の違いの差による．静的コンテンツの量が多いとその分プリフェッチできる量が増えるためである．なお，本提案機構を用いると，プリフェッチにより静的部分が先に表示されるため，ユーザの感じる待

表 2 事前処理の結果
Table 2 Result of preprocessing.

	グループ管理システム (group_kanri.php)	ショッピングシステム (goods.php)
事前処理前（php ファイル）	30.2 KB	7.5 KB
事前処理後（html ファイル）	1.3 KB	3.9 KB
事前処理後（php ファイル）	29.7 KB	6.9 KB

表 3 ロード時間の比較
Table 3 Comparison of loading time.

	グループ管理システム		ショッピングシステム	
	環境 1	環境 2	環境 1	環境 2
変換前	356 ms	1.26 s	540 ms	1.32 s
変換後	392 ms	1.21 s	417 ms	1.06 s
静的コンテンツの総容量	32.5 KB		136.3 KB	
ロード時間の変化	+10.1%	−4.0%	−22.8%	−19.7%



(a) 提案機構を利用せず

(b) 提案機構を利用

図 9 提案機構利用の効果
Fig. 9 Effect of using proposed approach.

ち時間は軽減されると考えられる．

また，両アプリケーションに対して，Cookie の読み込みや書き込み，文字列の置換部分にかかる時間を計測したところ，どちらも 1 ms に満たなかった．よって，本提案機構を適用したアプリケーションにかかるロスタイムとしては通信時間以外は考慮する必要はないと考えられる．

4.1.4 ロードの状況の違い

各種コンテンツのロードの状況を詳細に表示した結果を**図 9** に示す．結果は，Firebug [9] を用いて作成しており，クライアントがサーバと同一ネットワークにある場合のショッピングシステムの状況を示している．図 9 (a) は提案機構を利用しておらず，図 9 (b) は利用している．

まず，提案機構を利用していない場合（図 9 (a)）は元々の Web アプリケーションに対するアクセス状況である．この場合，ページ本体である goods.php ファイルがまずロードされる．そのあと，shop_title.gif や各種 jpg ファイルがロードされる．特徴として，goods.php ファイルがロードされた後，各画像ファイルをいっせいにロードしようとする．しかし，WWW サーバに 1 度につなげられるコネクション数が決まっているため，多くのファイルはコネク

ションに空きが出るまで待機（ブロッキング）状態となっている[*2]．最終的にロードが終了するまで 577 ms かかっている．

次に，提案機構を利用した場合（図 9 (b)），元の goods.php ファイルは goods.html（静的コンテンツ）と goods.php（動的コンテンツ）に分離されている．goods.html ロード後，提案機構が JavaScript コードを挿入する（jquery.min.js）．また，shop_title.gif も静的コンテンツとしてロードされる．以上の 3 つのファイル（goods.html，jquery.min.js，shop_title.gif）はキャッシュからロードされる[*3]．次に動的コンテンツを処理するために，goods.php が POST されるが，その結果として，各種 jpg ファイルを挿入する必要がある．これらのファイルはキャッシュに存在するため，キャッシュからロードされる．毎回サーバにアクセスする必要が生じないため，結果としてロード時間は，提案機構を利用していない場合と比べ，速くなっている．

上記の結果は，キャッシュにあるときの方がロードが速いことを示しているが，重要なポイントは，提案機構を利用した場合，ロードするファイル数が増えており，それが問題にならないということである．具体的には，提案機構を利用しない場合 goods.php の 1 つのファイルに対して，提案機構を利用した場合 goods.html，goods.php，jquery.min.js の 3 つのファイルが必要になる．

なお，図 9 (a)，(b) は 1 つの計測結果を示しているため，アクセスするたびに詳細な時間は異なるが，他の場合でも同様の傾向を示している．

4.2 静的コンテンツの容量の影響

前節の実験より，静的コンテンツが多ければ多いほど効果がありそうな結果となったが，本節ではそれを確かめる実験について述べる．

動的な Web アプリケーションのあるページに対して，意図的に静的コンテンツを増やした．そして，提案機構を利用しなかった場合と利用した場合（ただし静的コンテンツがキャッシュにプリフェッチ済み）のロード時間を比較した．

図 10 にその結果を示す．まず，提案機構を利用しなかった場合，静的コンテンツを増やすと，ロード時間が増えていく（図 10 “オリジナル”）．これに対して，提案機構を利用した場合，ロード時間は約 400 ms でほぼ一定である（図 10 “提案機構（全体）”）．提案機構を利用した場合，すでにプリフェッチされている静的コンテンツのロード時間は，キャッシュされている情報を表示するだけなので，静的コンテンツの容量が増えても約 200 ms 弱でほぼ一定

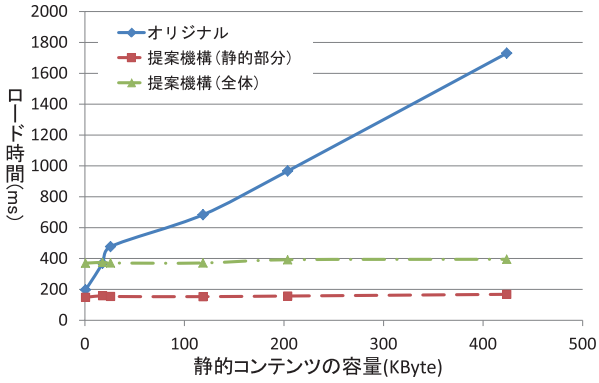


図 10 静的コンテンツの容量の影響
Fig. 10 Effect of size of static contents.

である（図 10 “提案機構（静的部分）”）．

静的コンテンツの容量が 20 KB よりも少ない場合，提案機構を利用しても効果がなく，むしろ提案機構を利用しない方が速い．しかし，20 KB よりも大きければ大きいほど提案機構の効果があることが分かる．

4.3 プリフェッチ対象の定期的更新

本提案機構では，アクセス状況を反映するように，定期的にアクセスログを解析し，アプリケーション管理者が指定した上位 i 件の静的コンテンツをプリフェッチ対象としている．本節ではこの 2 点について考察する．

まず，実施した実験では，データをとるため以外のアクセスはなかったため，途中でアクセスログおよび .htaccess ファイルの更新は行わなかった．.htaccess ファイルの更新頻度は，Web アプリケーション管理者が行うこととなるが，そのための 1 つの検討事項として，Web アプリケーションのコンテンツの更新頻度が考えられる．頻繁にページの更新が行われるようなアプリケーションでは，アクセス状況が変化しやすい．そのため，更新頻度を 1 日数回など頻繁に行った方がよいと考えられる．逆に，あまりページの更新を行わないようなアプリケーションの場合，（たとえば 3 日に 1 回など）あまり頻繁に .htaccess ファイルを更新しなくても大丈夫と考えられる．もちろんアクセスするユーザの種類に変化があると，アクセスするページも変わる可能性が十分ありうるため，最終的な判断は Web アプリケーションの管理者がアクセス状況を見ながら適宜更新頻度を調整する必要がある．更新頻度を設定する場合，どういう情報を利用し，どう行うべきかは今後の課題とする．

次に，プリフェッチ対象にすべき静的コンテンツ数については，トレードオフの問題である．静的コンテンツ数が多ければ多いほど，キャッシュヒット率が高くなり，ロード時間が短くなる．しかし，その反面，不要なコンテンツを増やすこととなり，ネットワークに対する負荷が上がることとなる．de la Ossa ら [10] の実験によると，トラフィックの増加が 5〜10%を超えると，ロード時間の削減率は減

[*2] 1326089470.jpg 以下のファイルでは灰色の部分があるが，これはブロッキング状態にあることを示している．
[*3] ステータス「304 Not Modified」はキャッシュからロードされることを意味する．

少し，プリフェッチの効果が低くなる．そこで，本提案機構も同様にトラフィックの増加が5〜10%以内になるようにプリフェッチ対象の数を調節すべきと考えられる．しかし，トラフィックの増加は単純なプリフェッチ対象数だけではなく，Webアプリケーションのアクセス頻度など様々な要因がある．詳細については，今後の課題とする．

4.4 本提案機構の利用制約

本提案機構の実装は，Expatというxml解析ツールを利用しているため，HTMLファイルが完全形である必要がある．ここで完全形とは，すべてのタグがXMLの階層に基づいて閉じられている状態を指す．また，現在の実装において，効果が得られないアプリケーションとして次のようなものが考えられる．

- 動的処理部分でprintやechoなどの出力関数を用いてすべての静的コンテンツを表示させてしまっている場合
- PHPのステートメントの最後のセミコロンが省略されている場合

本来は静的コンテンツであるはずのものの大部分が動的処理部分として記述されてしまっている場合，DCDによる静的コンテンツと動的処理部分の分離が不可能となる．よって，本提案機構の実装がうまく機能しないと考えられる．

また，PHPでは，スクリプトの末端部分である“?>”もステートメントの終端を表すことができるため，最後のセミコロンは省略することが可能である．しかし，本提案機構では動的処理部分を分離後，その末尾に「echo “<divide>”;」を挿入するため，スクリプトの末尾の位置が変わるのでエラーが起きてしまう．よって，PHPのステートメントの最後のセミコロンは省略してはならない．

5. 結論

本論文では，動的なWebアプリケーションに対応できるプリフェッチ機構を提案した．本機構では，動的なWebアプリケーションを静的と動的コンテンツの2つに分け，静的コンテンツ側をプリフェッチで取得し，動的コンテンツ側をAjaxで取得することで実現した．静的コンテンツのプリフェッチは，ユーザのアクセスを解析し，遷移確率の高いページを取得する方法を取った．

評価として，2つの既存アプリケーションに対して提案機構を適用した．静的コンテンツの容量が小さいアプリケーションに対しては提案機構の効果はあまりなかったが，静的コンテンツの容量が大きいアプリケーションに関しては，ロード時間が20%短くなった．

今後の課題としては，4.3節であげたものや4.4節であげた制約をなくすことがまずあるが，他に次のようなことがあげられる．

(1) PHPの動的処理部分に含まれてしまっている静的部分を分離する．アプリケーションによっては，本来静的コンテンツになりうるはずの部分もPHPの動的処理部分の中に含んでいる場合がありうる．よって，動的コンテンツ内のコードの解析を行い，完全な静的・動的コンテンツの分離を行うことで，ロード時間の短縮を望むことができる．

(2) 本提案機構では，プリフェッチのアルゴリズムとして非常に単純なものしか用いていない．アルゴリズムを選択可能にし，プリフェッチのパフォーマンスを上げることができれば，さらなるロード時間の短縮につながる．

(3) 長期間を要してプリフェッチのアルゴリズムの評価をとる．評価実験では，ロード時間のみに着目し，リファーラ・リクエスト表を用いなかった．そこで，リファーラ・リクエスト表を評価するためには，長期間をかけ，ユーザのアクセスログをもとにして，次に遷移するであろうページを予測する必要がある．既存研究としてde la Ossaがプリフェッチのアルゴリズムに関する評価を行っているので，本研究ではde la Ossaの評価を参照している[10]．しかし，対象とするアプリケーションや環境によって結果に差異が現れる可能性もあるので，プリフェッチのアルゴリズムの評価も今後の課題とする．

参考文献

[1] de la Ossa, B., Gil, J., Sahuquillo, J. and Pont, A.: Improving Web Prefetching by Making Predictions, *Proc. 3rd EuroNGI Conference on Next Generation Internet Networks*, pp.21–27 (2007).

[2] Linden, G.: Geeking with Greg (online), available from ⟨http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html⟩ (accessed 2012-05-14).

[3] Nielsen, J.: *Usability Engineering*, Morgan Kaufmann, San Francisco (1993).

[4] Padmanabhan, V. and Mogul, J.: Using Predictive Prefetching to Improve World Wide Web Latency, *ACM SIGCOMM Computer Communication Review*, Vol.26, No.3, pp.22–36 (1996).

[5] Dahlan, A. and Nishimura, T.: Implementation of Asynchronous Predictive Fetch to Improve the Performance of Ajax-Enabled Web Applications, *Proc. 10th International Conference on Information Integration and Web-based Applications and Services*, pp.345–350 (2008).

[6] Fisher, D. and Saksena, G.: Link Prefetching in Mozilla: A Server-Driven Approach, *Proc. 8th International Workshop on Web Content Caching and Distribution*, pp.283–291 (2004).

[7] The JQuery Foundation: jQuery (online), available from ⟨http://jquery.com/⟩ (accessed 2012-08-07).

[8] KOMONET: インターネットサービス KOMONET（オンライン），入手先 ⟨http://www.komonet.jp/⟩（参照2012-05-14）.

[9] Mozilla: Firebug – Web Development Evolved, Mozilla (online), available from ⟨http://getfirebug.com/⟩ (ac-

cessed 2012-05-14).
[10] de la Ossa, B., Gil, J., Sahuquillo, J. and Pont, A.: Referrer Graph: A Low-Cost Web Prediction Algorithm, *Proc. 2010 ACM Symposium on Applied Computing*, pp.831–838 (2010).

柴田 和祈

2010 年慶應義塾大学理工学部卒業．2012 年同大学大学院理工学研究科修士課程修了．同年ヤフー株式会社入社．Web アプリケーションに関心あり．

高田 眞吾 （正会員）

1990 年慶應義塾大学理工学部卒業．1992 年同大学大学院理工学研究科修士課程修了．1995 年同博士課程修了．博士（工学）．同年奈良先端科学技術大学院大学情報科学研究科助手．1999 年より慶應義塾大学理工学部情報工学科専任講師，現在，同大学准教授．ソフトウェア工学，情報検索等の研究に従事．電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE CS 各会会員．