HTML Templates that Fly

A Template Engine Approach to Automated Offloading from Server to Client

Michiaki Tatsubori Toyotaro Suzumura
IBM Research, Tokyo Research Laboratory
1623-14 Shimo-tsuruma, Yamato, Kanagawa, Japan
{mich,toyo}@jp.ibm.com

ABSTRACT

Web applications often use HTML templates to separate the webpage presentation from its underlying business logic and objects. This is now the de facto standard programming model for Web application development. This paper proposes a novel implementation for existing server-side template engines, FlyingTemplate, for (a) reduced bandwidth consumption in Web application servers, and (b) off-loading HTML generation tasks to Web clients. Instead of producing a fully-generated HTML page, the proposed template engine produces a skeletal script which includes only the dynamic values of the template parameters and the bootstrap code that runs on a Web browser at the client side. It retrieves a client-side template engine and the payload templates separately. With the goals of efficiency, implementation transparency, security, and standards compliance in mind, we developed FlyingTemplate with two design principles: effective browser cache usage, and reasonable compromises which restrict the template usage patterns and relax the security policies slightly but in a controllable way. This approach allows typical template-based Web applications to run effectively with FlyingTemplate. As an experiment, we tested the SPECweb2005 banking application using FlyingTemplate without any other modifications and saw throughput improvements from 1.6x to 2.0x in its best mode. In addition, FlyingTemplate can enforce compliance with a simple security policy, thus addressing the security problems of client-server partitioning in the Web environment.

Categories and Subject Descriptors

C.2.5 [**Local and Wide-Area Networks**]: Internet; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.3.2 [**Programming Languages**]: Specialized Application Languages

General Terms

Design Languages Performance Security

Keywords

Template engines, Client-server partitioning, Web applications

WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

1. INTRODUCTION

Web applications often implement the advantageous model-view-controller architecture [20, 11] by using *HTML templates*, often seeking to separate the webpage presentation from the business logic and objects. This is a mantra for experienced Web application developers conforming to that architecture [20, 4]. It has become a de facto standard programming model for Web application development [10, 5].

This paper proposes a novel implementation of server-side template engines that exploits the nature of the template-based programming model to enhance existing Web applications to boost their server throughput. Instead of producing fully-rendered HTML pages, the proposed template engine produces *skeletal scripts* that run on a Web browser on the client side. This bootstrap code includes only the template parameter values, which change with each request, and then retrieves the template data and the client-side template engine separately. This allows Web browsers to *cache the template data*, which is relatively static. Other skeletal scripts for the dynamic pages based on the same template can then share the cached template.

This architecture contributes to the reduction of server-side load since a server usually needs only to provide the skeletal scripts even though the dynamic data changes for each request. The improved server needs only to serve a client-side template engine for each new user and then a payload template for each newly visited type of page using the same template. These transmissions may also happen when a client-side cache is either stale or has a cache miss (because the website has evolved, or because files have been flushed out of the cache). Advantageously, the template engines and templates are reduced to static files on the server's file system, which makes it easier for a Web server to serve them compared to dynamically generating large and complex pages.

The differences in the implementations are almost *transparent* and most typical Web applications should run without any modifications except for replacement of the template engine. In fact, we made our prototype emulate the application programming interface of the Smarty template engine library [19] for the PHP language [21, 25], so it can replace the original Smarty libraries used in the SPECweb2005 application that we used for our performance tests. This simple method also works for at least some other Web applications such as SugarCRM.

While naive automatic client-server partitioning exposes *security vulnerabilities*, our implementation addresses these problems by using secure loading to the clients. This in-

volves the automatic enforcement of a simple security policy controlled by the administrator of the Web application. Security is always of concern in publicly available Web applications [29]. Typical automated client-server partitioning technologies such as Hilda [28] have ignored the security problems caused by porting some parts of the server-side logic of a Web application to the untrusted clients.

The contributions of the paper are:

- a proposal for an template engine approach to an automated client-server partitioning system compatible with the existing Web architecture
- a design and implementation with efficient cache usage and security
- experimental results for the proposed template engine with an application in SPECweb2005, which is an industry standard benchmark
- potential promotion of template-based Web application development with the additional advantage of automatic performance gains from using the proposed template engine

The rest of the paper is organized as follows: In Section 2 we motivate our proposal by discussing the programming model and implementation of a reference template engine, used as a representative of current template engines. Section 3 introduces FlyingTemplate, our proposal for the design of a novel server-side template engine. In Sections 4 and 5, we discuss the important implementation issues affecting efficiency and security. We report on our experimental results in Section 6. After discussing related work in Section 7, we conclude the paper in Section 8.

2. TEMPLATE-BASED PROGRAMMING

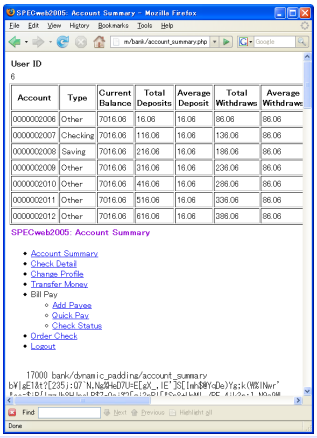
Template-based Web programming is popular mainly because it separates the page representation, the "views", from the business logic and data of a program, the "controls and models". Such templates are available as software libraries [20, 11], as programming or modeling language features [5, 2] or as Web application frameworks [10, 4]. The benefits include encapsulating the look and feel of a website, clearly described views, a better division of labor between graphics designers and coders, component reuse for view designs, unified control over the evolution of the appearance, better maintainability of the runtime, interchangeable view artifacts for different development projects, and security compatible with end-user customizability [20].

2.1 Template Usage

In general, there are three steps in using a template engine:

- Specifying a template to use,
- Assigning values to the parameters as the actual content, and
- Filling in the template with the assigned values to obtain the HTML results.

Though the application programming interfaces may vary for each template engine, they are essentially built around these steps.




```
check_login();
$smarty=new SmartyBank;

$summary=backend_get($_SESSION['userid']);

$smarty->assign('userid', $_SESSION['userid']);
$smarty->assign('summary', $summary);
$smarty->display('account_summary.tpl');
```

Figure 1: A final end-user view and a simplified PHP script using a template engine.

In this paper, we use the Smarty template engine library [19] for the PHP language [21, 25] as the reference implementation of a template engine. This library is used in many production-quality open source Web applications such as XOOPS and SugarCRM. It is even used in SPECweb2005 [22, 26], the industry-standard Web server benchmark kit, with available implementations for both PHP and Java. We used the release version 2.6.7 of Smarty, which is distributed with the SPECweb2005 environment that we used for our experiments.

Figure 1 includes a simple PHP script written using Smarty, extracted from SPECweb2005 and greatly simplified for this explanation. This code first authenticates and authorizes the user by calling a user function `check_login()`. Then it instantiates a template engine object (from the class `SmartyBank`) and assigns that reference to the variable `$smarty`. Next it calls the user function `backend_get()` to access the backend database for the required user data. The `$_SESSION` is a globally accessible associative array variable that holds the session data for the user. Here, the `userid` is a key associated with a user identifier number. The resulting data is stored into the variable `$summary`. Finally, it renders the output result based for the user ID with the data obtained from the backend using the template. The PHP runtime passes the output result to the frontend Web server to be included in its HTTP response.

In this example, the usage process of the Smarty template engine essentially consists of the calls to two methods, `assign()` and `display()`. The `assign()` method is called to assign new values to the parameters, while the `display()` method is called both to specify a template and to fill it with the assigned values. For the example in Figure 1, the template parameter `'userid'` is bound to the user ID while the other parameter `'summary'` is bound to the results constructed from the data obtained from the backend database.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" ..>
<html>
 <head><title>SPECweb2005: Account Summary</title></head>
 <body bgcolor="white">

 <table summary="SPECweb2005_User_Id">
  <tr><th>User ID</th></tr>
  <tr><td>{$userid}</td></tr>
 </table>

 <table summary="SPECweb2005_Acct_Summary" cellpadding=3 border=1>
  <tr><th>Account</th><th>Type</th><th>Current Balance</th>…</tr>
  {foreach item=acct from=$summary}
    <tr>
    <td>{$acct[0]}</td>
    <td>{if $acct[1] eq "1"}
          Checking
        {elseif $acct[1] eq "2"}
          Saving
        {else}
          Other
        {/if}</td>
    <td>{$acct[2]}</td>
    …… parameterized burdens for other columns ……
    </tr>
  {/foreach}
 </table>
 …… other burdens ……
 </body>
</html>
```

Figure 2: An example template taking two parameters `$userid` and `$summary`.

The name of the template file `'account_summary.tpl'` is specified when generating the final result.

2.2 Expressive Power of a Template Language

The syntax used in templates is usually not limited to language constructs for creating "holes" in an HTML document with simple expressions to embed references to parameter values. The syntax available to describe a typical template is much richer. It may include iterators over array values, selections based on parameters, and references to other templates. In addition, there are often language constructs for calculations based on one or more parameter values and literal constant values, and usage of template local parameters for storing intermediate results.

While a template language is just a programming language, templates make sense only if they are used for the sake of simplicity that would be negated by complicated embedded logic. For example, even PHP can be regarded as a template language from a certain perspective, since the programmer embeds PHP code in an HTML document. In fact, in practice most of the template engines such as Smarty offer Turing-complete languages, and they can easily lose their advantages when they are used without regard for certain rules about the separation of concerns [20, 4].

Complex but still modest usage of language constructs appears in SPECweb2005. Figure 2 is a Smarty template used for the account summary page of the SPECweb2005 banking application (also greatly simplified from the original for this explanation). The parts in bold are embedded "programming" and the other parts are HTML text that will be copied out just as it appears. The embedded directive `{$userid}` refers to the template parameter `$userid` to dump a parameter value into that position.

The directive `{foreach ..}`, paired with `{/foreach}`, indicates iterating over the elements of the array value of the template parameter `$summary`. It generates an HTML page fragment from the other template between the markers for each element, while assigning each element to a new local variable `$acct`. Since the variable `$acct` still holds each element of the account information as an associative array value, its array elements are accessed through the form of `$acct[n]`, where `n` is an associative key. This template also uses the selection language constructs `if` and `elseif` while computing Boolean values using the equality test operator `eq`.

3. FLYING TEMPLATE

FlyingTemplate is a server-side template engine. It looks much like a regular server-side engine that does the template-filling work on the server side, but actually lets the clients do that work. In this paper, we use Smarty as a reference template engine and PHP as the underlying programming language, but the design of FlyingTemplate itself should be applicable to other template engines and programming languages.

3.1 Goals & Design Principles

The major design goals of FlyingTemplate are:

Efficiency – FlyingTemplate should perform better than existing template engines, at least in typical circumstances.

Standards compliance – The implementation should conform to Web standards.

Implementation Transparency – Existing applications should run correctly without modifications.

Server Security – Introducing FlyingTemplate should not create unexpected security vulnerabilities.

The first goal, efficiency, is essential since FlyingTemplate makes sense only if it can improve the Web application compared to its original template engine. FlyingTemplate should not change the visible functionality of a Web application, but enhance the server throughput, which is the primary reason to use FlyingTemplate. The other three goals are for adapting existing Web applications to work without modification in the same environments, and to allow programmers to work with existing template-based programming models. Compromising on the later goals may make sense for FlyingTemplate, but would limit its applications.

Therefore, for FlyingTemplate to have a broad appeal, the last three goals are also important. FlyingTemplate needs to conform to the standards in order to handle most of the current Web architectures and existing Web applications that depending on the standards. That includes formal standards such as the HTTP protocol and de facto standards such as the major browser implementations. It is best if we can apply FlyingTemplate to existing applications without changing their code since that work would involve extra coding work and possibly create bugs, and might sometimes be impossible for inaccessible application code. Finally, security is important when considering publicly accessible applications in the Web environment, where we cannot trust the client platforms.

To achieve the goals, we developed FlyingTemplate with these design principles:

Effective cache usage – Leveraging the browsers' caches is the best way to reduce the load carried by the server for the clients.

```
<script src="/lib/filler.js"></script>
<script>
fill_template("account_summary.tpl",
[["userid","6"],
 ["summary",
  [["0000002006","00","7016.06","16.06","16.06","86.06","86.06"],
   ["0000002007","01","7016.06","116.06","16.06","136.06","86.06"],
   ["0000002008","02","7016.06","216.06","16.06","186.06","86.06"],
   ["0000002009","03","7016.06","316.06","16.06","236.06","86.06"],
   ["0000002010","04","7016.06","416.06","16.06","286.06","86.06"],
   ["0000002011","05","7016.06","516.06","16.06","336.06","86.06"],
   ["0000002012","06","7016.06","616.06","16.06","386.06","86.06"]]]]);
</script>
```

Figure 3: A skeletal result including bootstrap JavaScript code.

Reasonable compromise in transparency – Instead of making the implementation completely transparent, we focus on the typical uses of templates and rely on simple security policies that are acceptable for most Web applications.

In the rest of this section, we will describe the basic ideas of the design and architecture of FlyingTemplate. The discussions of cache use and security appear in Section 4 and Section 5.

3.2 Skeletal Results

FlyingTemplate generates tiny pieces of client-side scripts instead of generating fully rendered HTML documents. An example of such code is shown in Figure 3, which is JavaScript code within HTML. This bootstrap code has two components:

- a client-side template engine loader
- a template engine invocation with a template ID and template parameter values embedded in the code

In the example, filler.js is a JavaScript supplied by the original server, and the expression `fill_template()` calls a function for the template application. The first argument "account_summary.tpl" is a template ID as a string literal, while the second argument is a literal construction for an associative array which maps each parameter name to its value.

The client-side template engine uses two functionalities, XMLHttpRequest (XHR) and Dynamic HTML (DHTML), which are available for scripts running on recent Web browsers. These functionalities are known as the basis for Ajax-style programming [12] and are widely available in popular browser implementations such as Mozilla Firefox, Microsoft Internet Explorer, Opera, Apple Safari, and Google Chrome. XHR allows client-side scripts to asynchronously access the original server via an HTTP connection [27]. DHTML allows scripting languages to change the HTML constructs in a webpage, which in turn affects the look and function of the otherwise-static HTML content, after or as the page is loaded [13].

The bootstrap code retrieves the template data, in addition to the client-side template engine. As illustrated in Figure 4, a client needs additional interactions with the original server in the new architecture, compared to the original interactions between the Web application server with a traditional template engine and its clients. The bootstrap code fetches and loads template engine scripts using DHTML, by dynamically adding new `<script>` tags with `src` attributes specifying the URL of the engine scripts. It also retrieves

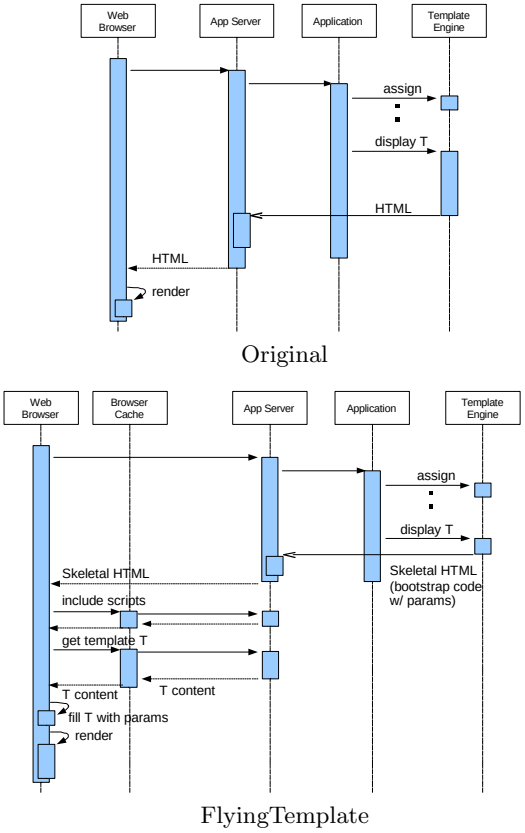


Figure 4: Interaction sequences with a normal template engine (top) and the FlyingTemplate template engine (bottom).

```
function display($template_id) {
  $js_template = "\"{$template_id}\"";
  $js_params = json_encode($this->params);
  echo '<script src="/lib/filler.js"></script>';  // template engine
  echo "<script>fill_template({$js_template},{$js_params});</script>";
}
```

Figure 5: Server-side code fragments.

a template by using XHR. Finally the loaded template engine generates an HTML document object according to the template and parameter values, inserting the results into the HTML content rendered in the browser's graphical user interface.

3.3 Server-Side Template Engine

The server-side implementation of FlyingTemplate emulates the original template engine of the Web application. It provides the same usability for the template engine as discussed in the section 2.1. For Smarty [19], we needed to provide the application programming interfaces for `assign()` and `display()`, as shown in Figure 1.

The core implementation of the logic is the generation of client-side code as shown in Figure 5. We omit the implementation of `assign()` since this is just the code for storing the given key-value pair in a table in the template engine in-

stance. This implementation is unchanged from the original Smarty.

The code generation algorithm is also fairly simple. In order to generate the bootstrap code shown in Figure 3, the engine first provides an embedded representation of the template ID and parameter values. In PHP, there is a utility function `json_encode()` for converting PHP objects into JavaScript literal representations in the JSON format [16]. The code generator then emits the `<script>` tags and template invocation code supplied filled in with the values provided with the template ID and parameter values.

While it is omitted in Figure 5, there could be a test and branch in the body of `display()`. The test is for examine whether the template is written using a limited set of expressions that the template engine can safely and efficiently replace with skeletal results. The set of expressions and the method of analysis can vary for each implementation, but that topic is beyond the scope of this paper. However, such an implementation could cache the analytic results to greatly reduce the costs of testing.

4. PRECISE CACHE INVALIDATION

The templates are static content most of the time, so we would like to cache them on the client-side whenever possible. However, we should not assume that they never change as their Web application evolves. At the same time, effectively exploiting caching is crucial to realize the benefits of FlyingTemplate. Unless the templates are cached at the client's side, the Web browser would always fetch the templates from the server as well as the skeletal results, which would increase the amount of transfers over the original reference template engine. While there could remain the advantage of offloading HTML document generation task, the benefit of reducing the network bandwidth consumption would completely disappear.

Browser caches and their cache validation mechanisms using the HTTP If-Modified-Since request-header can partially address this problem. The If-Modified-Since request-header field makes the response conditional: if the requested variant has not been modified since the date of the cached data retrieval, no actual content but a 304 (not modified) response will be returned [15] to reduce the size of the response message. However, pairs of request and response are still required with this mechanism, even for cache entries that are quite fresh.

In order to reduce the extra requests and responses for fresh cache entries, we should first *suppress cache validation*, and then *trigger revalidation on demand* by using the client scripts to control the state of the cache. The rest of this section discusses how to implement them based on the HTTP protocol.

4.1 Suppressing Cache Validation

We can suppress cache validation by specifying "never expires" as the term of validity for the content returned from a server. Methods for specifying the term of content validity are available in the HTTP protocol. The Expires response header has been available since HTTP/1.0 and the Cache-Control response header, which was introduced in HTTP/1.1, also allows use of the "max-age" directive. We can specify one year as the term, which is the maximum time allowed by the specification and sufficiently long for our purposes.

In addition, we could potentially leverage heuristic freshness, in which a client can guess a cache entry's plausible validity from its last modified time and last accessed time. It is allowed by the HTTP specification to assume a cache entry is fresh if the elapsed time since the last access is less than some ratio, say up to 10%, of the elapsed time since the last modified time. Since the computation of the expiration period is based on the age of the content, we could set the last modified time of the content to be much earlier to make the expiration time longer.

4.2 Stale Template Notification

It is not a good idea for a server to try to track the states of the cached templates on numerous clients. Theoretically, it would be possible for a server to remember the versions of the templates served to each client. In this approach, the server could notify each client about any template changes when returning a skeletal result with a template ID. However, this approach would require the server to maintain a significant amount of client state data, which would have a negative impact on server scalability.

The server should, instead, simply include the *version* information of the specified template when returning the skeletal result. The version information of each template is available from the cache. It can be basic information appended to the original template, but we recommend leveraging the Last-Modified field of the cached HTTP response for the template. XHR offers a programming interface, `getResponseHeader()` [27], for handling the HTTP headers of responses, even if they are being retrieved from the cache. Since the date and time in the field was originally specified by the server, that information is known when sending skeletal results.

An alternative to using the Last-Modified header would be to use the ETag header available in the HTTP protocol. An HTTP client may use the ETag response header field for comparison with other entities from the same resource. The server can compute the ETag field value, which is usually based on the file name and modification time. The use of the ETag header field values has a potential advantage over the use of the Last-Modified header since this header is provided specifically for such purposes. We will discuss this again in Section 4.4.

It is possible for the server to compute these values with limited CPU resources. In general, constructing the Last-Modified or ETag field values is not free since it requires retrieving, formatting, and encoding the modification time of a file into a serialized form of the value. However, the computation result can easily be cached at the server side, similarly to the logic of caching the compiled templates at the server side in a traditional template engine such as Smarty [19]. The heavy computation is only required when a newly modified template is first accessed on the server.

4.3 Validation on Demand

A client script first needs to determine whether or not a cached entry is fresh, based on a template ID and its version information such as a Last-Modified field value or an ETag field value, which it received as a part of a skeletal result from the server. To retrieve the cached entry, it should just issue an XHR call for the specified template on the server. The response for this request is obtained from the cache, since the expiration time is "never expires", as already men-

tioned. Then the client script can obtain the Last-Modified field or the ETag field from the cached template response using `getResponseHeader()` to compare it with the value given in the skeletal result.

Then, the client may request an *end-to-end reload* or an *end-to-end revalidation* according to these specifications [15], which are defined as:

End-to-end reload. The request includes a "no-cache" cache-control directive or, for compatibility with HTTP/1.0 clients, a "Pragma: no-cache". The server MUST NOT use a cached copy when responding to such a request.

Specific or unspecified end-to-end revalidation. The request includes a "max-age=0" cache-control directive, which forces each cache along the path to the origin server to revalidate its own entry, if any, with the next cache or server. The revalidation is specific if the initial request includes a cache-validating conditional with the client's current validator, otherwise called unspecified.

XHR offers a programming interface `setRequestHeader()` to add to the HTTP headers of requests.

4.4 Consolidated Invalidation

The combination of the ETag response header and the If-None-Match request headers can simplify the validation process. While the original process still requires the two steps of examining the freshness of the cache and revalidation of a stale cache, we can consolidate the examination step with the revalidation step by specifying an ETag field value in the If-None-Match field.

According to the HTTP specification [15], the ETag response-header field provides the current value of the entity tag for the requested variant. A server and its clients may use the entity tag for comparison with other entities from the same resource. Then a client may use the If-None-Match request-header field to make the method like GET act conditionally. A client that has one or more entities previously obtained from the resource can verify that none of those entities is current by including a list of their associated entity tags in the If-None-Match header field. The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead.

4.5 Ajax Caveat in Practice

Despite the useful features of HTTP/1.1, as is often the case with Ajax technologies, the browser implementation of caches for XHR is immature and sometimes lacks the features required to implement this revalidation process. Table 1 summarizes the current implementation status of popular Web browsers in terms of efficient XHR cache implementations with HTTP/1.1 conformance [1]. We tested the latest stable releases of five popular Web browsers on Windows XP SP2, as of October, 2008: Mozilla Firefox 3.0.3, Microsoft Internet Explorer 6.0 SP2, Opera Web Browser 9.61, Apple Safari 3.1.2, and Google Chrome 0.2.1.49.

All of the tested browsers except Opera supported HTTP response caching for XHR. Opera is very conservative (marked "cons." in the table) in conforming to the HTTP specification, but inefficient since caching is never used. The other 4 browsers cache effectively, while sometimes failing to conform to the HTTP specification (marked "fail" in the table). Since we cannot do anything with Opera as regards caching, we will only consider the other browsers for the rest of this section. In other words, FlyingTemplate will not work effectively when the tested version of the Opera Web Browser is used for browsing a Web application.

Fortunately, the other 4 browsers properly consider the value of the Expires response-header field and the Cache-Control max-age directive when processing XHRs using their caches. While not of much concern in this paper, it is known that there are differences in how and when the cached data is revalidated when a human user indirectly requests revalidation, according to the cache testing website we used. Firefox revalidates the cached response every time the page is refreshed, issuing an "If-Modified-Since" header with the value set to the value of the "Last-Modified" header of the cached response. Internet Explorer does so only if the cached response is expired. The performance of the FlyingTemplate approach suffers from users frequently refreshing the page, as by hitting the "reload" button.

However the revalidation of cached entries with "never expires" seemed problematic. First, none of the browsers handled the most convenient If-None-Match header effectively. They all tried to load from the original content whenever the If-None-Match header was used. In addition, some of these browsers do not respect the no-cache or max-age=0 directives, which are used for forcing revalidations of unexpired cache entries that a client script knows to be stale.

A workaround for these problems is to use the If-None-Match header for refreshing the stale cache entries. The script should first examine the cache's freshness and then force stale cache data to be refreshed using the implementation side effects of the caching behavior for If-None-Match requests.

5. SECURITY

In general, running some parts of a Web application which has been developed to run on server side on the client side creates security vulnerabilities. There are especially severe risks for the server as regards the confidentiality of the server data, the integrity of the inputs for server data, and the accessibility authentication. The Web application of Figure 1 described in Section 2 offers an example. Running the logic of the original PHP code from the server side on the clients could allow an altered Web browser to access the backend DB directly through `backend_get()` as an arbitrary user, thus skipping the `check_login()` (an authentication problem). Even for an authenticated user, this vulnerability might allow accessing the backend DB directly to obtain the data that was originally filtered out (a confidentiality problem) or altering the data (an integrity problem).

While it is an interesting research theme as how to retain the semantics of the original Web application including its security, we slightly compromised the security for FlyingTemplate. Instead, we are focusing on achieving both an efficient implementation of the Web applications and acceptable security with a simple security policy that developpers can easily understand and specify. The security policy is simply that:

[1]The browser cache tests in this paper are conducted based on http://www.mnot.net/javascript/xmlhttprequest-/cache.html while we added some of missing tests for our purpose.

Table 1: Browser cache tests for XHR calls.

Tests			Firefox 3.0.3	MSIE 6.0SP2	Opera 9.61	Safari 3.1.2	Chrome 0.2.149	Annotation
Request Cache-Control	Header Pass-Through	Cache-Control: no-cache honored	fail	fail	cons.	fail	success	cons. (Not cached), fail (Cache-Control: no-cache not honored)
		Cache-Control: max-age=0 honored	fail	fail	cons.	success	success	cons. (Not cached), fail (Cache-Control: max-age=0 not honored)
		Cache-Control: only-if-cached honored	fail	fail	cons.	fail	fail	cons. (Not cached), fail (Cache-Control: only-if-cached not honored)
		Pragma: no-cache honored	fail	fail	cons.	fail	success	cons. (Not cached), fail (Pragma: no-cache not honored)
	Automatic Cache-Control	Cache-Control not automatically appended	success	success	success	success	success	fail (Request cache-control header sent)
Validation	Conditional Headers	If-Modified-Since request header used	success	success	fail	success	success	fail (Header not sent)
		If-None-Match request header used	cons.	cons.	fail	cons.	cons.	cons. (Fresh representation not used), fail (Header not sent)
	304 Not Modified Handling	304 status code automatically handled	success	success	fail	success	success	fail (304 not handled)
Freshness	Basic Freshness	Heuristic freshness	success	success	cons.	success	success	cons. (Freshness heuristics not used)
		Cache-Control max-age response freshness	success	success	cons.	success	success	cons. (Fresh representation not used)
		Expires response freshness	success	success	cons.	success	success	cons. (Fresh representation not used)
	Query Freshness	Heuristic freshness w/ "?"	fail	fail	success	fail	fail	fail (Stale representation used)
		Cache-Control max-age response freshness w/ "?"	success	success	cons.	success	success	cons. (Fresh representation not used)
		Expires response freshness w/ "?"	success	success	cons.	success	success	cons. (Fresh representation not used)
	Private Caches	Private responses cached	success	success	cons.	success	success	cons. (Not cached)
	Variant Caching	Arbitrary Negotiated response caching	cons.	fail	fail	cons.	cons.	cons. (Fresh representation not used), fail (Not used)
		Arbitrary Negotiated response differentiation	success	success	success	fail	success	fail (Negotiated response used when it shouldn't have been)

- Template data cannot be confidential.

We believe this security policy is reasonably acceptable for most Web applications because of the nature of template-based programming for the separation of views from models and logic. The designer of the view of a page should not be concerned about the server-side security of the Web application. In fact, we found many Web applications using Smarty do place the template data at publicly accessible locations with file names that can easily be guessed from the original URLs of the dynamically generated pages using those templates.

6. PERFORMANCE EVALUATION

This section evaluates our approach by running a standard Web benchmark application, the Banking scenario of SPECweb2005, and also gives a detailed analysis of the performance results.

6.1 Experimental Environment

We use Lighttpd 1.4.19 as the server, running on an IBM IntelliStation M Pro with a single 3.4 GHz Xeon processor and 2 GB of RAM running Fedora Core 7 (kernel 2.6.23). Lighttpd was configured to use 1 parent process and 2 worker processes, and 8 FastCGI processes. For the client machines, we use an IBM IntelliStation M Pro with a single 2.0 GHz processor machine connected to the server via a 1-GB Ethernet LAN. The network latency was 0.12 ms, and the actual network throughput measured by netperf was 941 Mbit per second. For the PHP runtime, we used PHP 5.2.6 with APC (Alternative PHP Cache) enabled to eliminate the parsing overhead. To measure the server throughput, we used Apache JMeter 2.3.2. We tuned up the environment carefully based on our prior experience [26].

6.2 Performance with SPECweb2005

To show the validity of our approach, we used SPECweb-2005, which is a standard Web application benchmark defined by SPEC. The three-tier architecture used for SPECweb-2005 consists of a Web server, a PHP runtime, and a BESIM simulator which simulates database behavior. The banking scenario represents an online banking Web application, characterized as a secure Web application with SSL communication, such as checking account information, transferring money to different accounts, and so forth. The scenario is comprised of 12 different web pages, and Table 2 shows comparisons of the average data size for each webpage in the banking scenario for the original approach and for our approach. The table clearly shows that the data size to be transferred is greatly reduced.

Table 2: Average data size transferred for each page.

(bytes)	Original	FlyingTemplate
account_summary	18195	361
check_detail_html	12731	271
profile	34295	327
change_profile	24566	289
transfer	30393	255
post_transfer	15067	283
place_check_order	26438	288
order_check	25678	312
bill_pay_status_output	23584	378
quick_pay	17821	294
bill_pay	35400	270
post_payee	20136	208
add_payee	17781	327

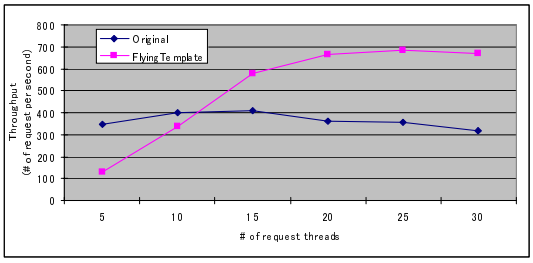


Figure 7: Throughput with varying number of client threads.

Figure 6 illustrates the performance results of the original and our proposed runtime configuration. The vertical axis shows the throughput as the average number of requests per second for each webpage indicated on the horizontal axis. The graph compares different loads with varying number of clients, 1, 2, and 4 clients. Each client, corresponding to one JMeter load generator process, invokes 24 simultaneous requests with Java threads. As shown in the graph, our approach outperforms the original configuration by at least 59% in the post_transfer page, and by a maximum of 104% in the profile page when compared with 1 client.

Figure 7 illustrates the throughput with varying numbers of simultaneous requests from 10 to 30 from each client. This experiment was performed in order to understand the appropriate numbers of requests to achieve the best throughput, since the previous experimental results were all obtained

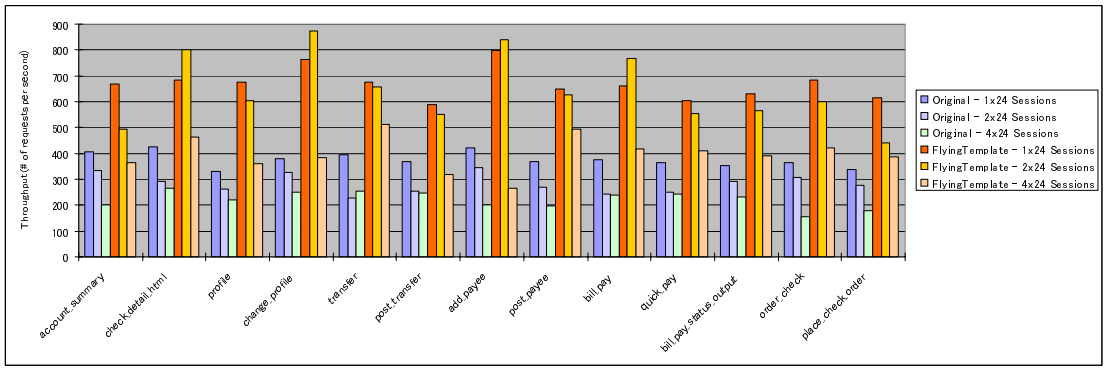


Figure 6: Throughput for a SPECweb2005 Banking scenario.

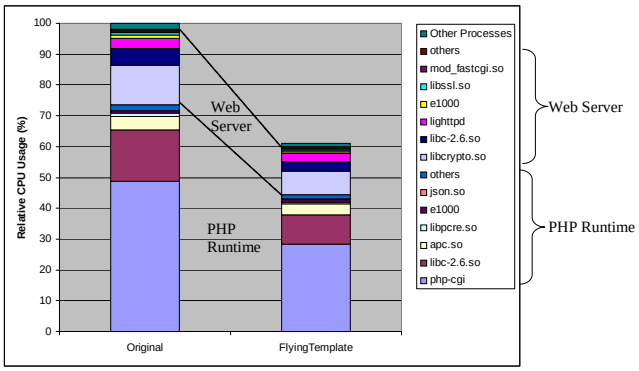


Figure 8: Relative CPU usage for processing a single request.

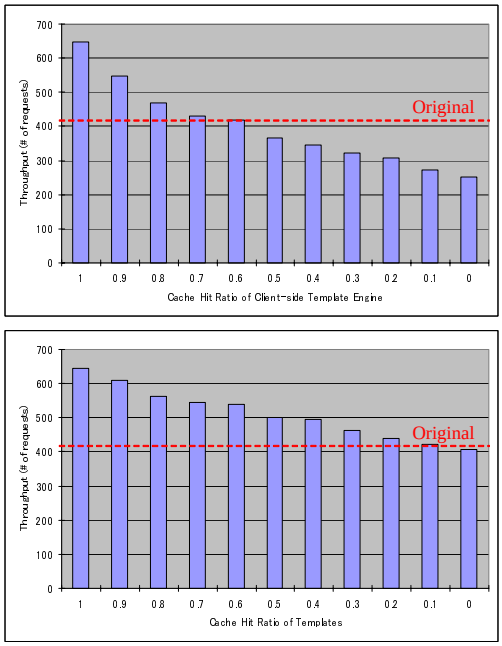


Figure 9: Throughput for various virtual cache hit ratios of client-side template engine (top) and templates (bottom).

with the limited configuration of 24, 48 and 96 simultaneous requests. The figure demonstrates that there is not much difference in throughput for various numbers of client threads with the moderately high loads from 20 to 30 threads, for each template engine configuration. With moderately high load, the CPU usage is close to 100%. Under lower loads, the response times of the two configurations are at their best, but the performance differences are below the noticeable levels for human users, and thus do not matter.

6.3 Profiling Result

Figure 8 shows the relative CPU usage in processing a single request for the account summary page in the SPECweb2005 Banking scenario. As shown in this figure, the sharing of CPU usage required for each component is similar between the two configurations. The JSON component, which is an additional component required for FlyingTemplate, consumes only a small fraction of the CPU time. In addition, the figure shows that our approach reduces the CPU usage for SSL processing (libcrypto.so in the graph). This result was measured by the profiling tool called oprofile, and is normalized by the relative throughput results provided by the previous experiment.

6.4 Cache Effect

Our approach assumes that all of the template engine operations are performed on the browser side and that the related files are cached most of the time. The two kinds of files, the client-side template engine files and the templates, are designed to have different characteristics for their cache hit ratios. On the one hand, a single template engine can usually be shared among any pages within a Web application or a website. The cache misses for the client-side template engine always occur when the user of the Web application first visits the website, or may occur when the cache is cleared at the client side or when the Web application changes the template engine being used. In contrast, the templates usually differ for each type of dynamically generated page. For example, the template for the account summary pages in the SPECweb2005 Banking scenario is different from the templates for other pages such as profile pages in the same scenario. Therefore the cache misses for a template are likely to be more frequent than misses on the template engine and may also occur for the same kinds of reasons of client-side cache clearance or Web application change.

To understand the effects of the cache, we compared the server throughput with the varying virtual cache ratios shown in Figure 9. This experiment assumes variations in cache hit probability for the template engine and templates. First, we used various virtual cache hit ratios for the client-side template engine, $P(E)$ (E: event of template engine cache hit), while holding the ratio for templates, $P(T|E) = 1$ and $P(T|E') = 0$ (T: event of template cache hit), as shown in the top graph. Then we used various virtual cache hit ratios for the templates, $P(T|E)$, while holding the ratio for the template engine, $P(E) = 1$, as shown in the bottom graph.

The top graph of Figure 9 shows significant throughput degradation because of the frequent misses for the template engine cache. In the experimental results, the worst case with the template engine cache hit ratio of 0 shows a 61% performance reduction compared to the ideal case where the cache hit ratio is 1. The throughput was worse than the original template engine configuration when the cache hit ratio was lower than 0.6, which we believe unlikely to occur for typical Web application scenarios. The significant degradation is due to the large size of the client-side template engine scripts. The client-side template engine for the experimented FlyingTemplate implementation consists of three main JavaScript files (about 33 kB in total), with one or more small JavaScript files (less than 1 kB) depending on the template engine functions required for each page.

The bottom graph of Figure 9 shows modest throughput degradation as the template cache misses increase. The situation is far better than the case for template engine cache misses. In fact, the throughput is not as bad as the original template engine configuration even when the template cache misses every time. This occured because the separate downloading of files avoids copying and processing large template files in the PHP engine. The sizes of the transferred templates are similar to the transferred data sizes for the original configuration in Table 2. For example, the total size of a template and a file called dynamic_padding included for the template in the account summary page of SPECweb2005 is about 18 kB.

7. RELATED WORK

In the context of the distributed computing research, we can regard FlyingTemplate as one of the automatic partitioning systems for a Web environment similar to Hilda [28]. However, fully automated partitioning of a normal program originally written assuming to work at server side poses security problems in the Web environment as we discussed in Section 5.

Security

As far as the authors know, Swift [6] is the only work proposing an automatic client-server partitioning system in a Web environment with strong consideration of security. By forcing programmers to write security annotations, it automatically partitions the programs written in a special but uniform language to run as client-side JavaScript and server-side Java. According to the security annotations, security-critical code and data are always held on the server. The efficiency of the partitioned system depends on the complex analysis of the entire program, which may not always be precisely optimal. The approach of FlyingTemplate can be regarded as greatly easing this kind of security annotation task and giving heuristics for efficient partitioning according to the convention of the template-based programming model.

Function Shipping

Automated partitioning means the partitioning should be transparent to the application programs and application programmers. This transparency characteristic differentiates it from Web-based mobile agents [9], since that approach requires programmers to write code according to their frameworks and programming models.

The idea of transparently moving some functions in existing software to the hosts suitable for their execution has been studied and examined for several decades. Significant amounts of work have been done, especially on the requirements of the host computers [7] and clusters [1]. More recent studies have explored techniques for automatically partitioning legacy applications to multiple hosts for functional distribution according to given policies [23, 24]. Generally applying these techniques in a Web environment needs to solve the same, efficiency and security issues, and those problems are still open.

Leff et al., used a model-driven approach in applying the model/view/controller design patterns to Web applications in a partition-independent manner [18]. The applications were developed and tested in a single address-space but his approach allows deploying them to various client/server architectures without changing the applications' source code. Since the partitioning decisions can be changed without modifying the application, a designer can treat security policy as a partitioning decision. It is, however, difficult to link partitions to their security vulnerabilities.

Client-side Templates

Style sheets and XSLT can be regarded as templates for HTML and XML documents. In particular, XSLT offers a great expressiveness, allowing a translator to produce complex HTML documents from dynamically generated XML elements. While this approach requires mastering two quite different languages, programmers with the ability to write both the server-side generator of the XML elements and the appropriate XSLT templates can automatically benefit from the client-server partitioning. Actually, we implemented another version of FlyingTemplate that generated XML documents associated with XSLT style files as skeletal code. We were able to generate these XSLT files from the original templates, but it was very difficult to write XSLT translators generating ill-formed HTML documents. This happened quite often with existing Web applications, including SPECweb2005.

Our current implementation of FlyingTemplate is highly dependent on the implementation of the client browsers, and the specific techniques and technologies known as Ajax. The essential part of the client-side capabilities are client-side scripting capabilities that can communicate with the original server and dynamically change the structure of their HTML documents as rendered on the browser [14, 8].

XMLHttpRequest [27] is one of the most famous communication methods, and implements an interface exposed by a scripting engine that allows scripts to perform HTTP client functions, such as submitting form data or loading data from a server. Other alternatives are available, but with various benefits and drawbacks, such as iFrame calls [17], image-cookie calls [3], etc.

8. CONCLUDING REMARKS

In this paper we proposed FlyingTemplate, which is a server-side template engine that automatically transfers more of the task of generating HTML documents to the client browsers. Instead of producing a fully-generated HTML page, the proposed template engine produces a skeletal script which includes only the dynamic values of the template parameters and the bootstrap code that runs on a Web browser at the client side. We designed the architecture of the client-server partitioning for effective browser cache use with the enforcement of a simple server security policy. A prototype implementation can successfully be substituted for an existing server-side template engine in a SPECweb2005 application with from 1.6x to 2.0x improvement of the server throughput at peak.

While there are limitations in the programming style of templates when taking advantage of our FlyingTemplate, we believe such limitations are acceptable when using templates for separation of views from controls and models in a Web application. Our prototype and experimental results also imply a potential, additional benefit of template-based Web application development. As an Ajax application, it also anticipates reasonable demands for the standardization of Web browser cache behavior and implementation conformance.

9. ACKNOWLEDGMENTS

Members of the Dynamic Scripting Language group in IBM Tokyo Research Laboratory, to which the authors belong, worked togather on PHP runtime research as a team, and thus have influenced over the work presented in the paper from various perspectives. Especially, Scott Trent worked very hard on experimental environment set-up which we also used in our experiments in the paper. We also thank Akihiko Tozawa and Tamiya Onodera for joining in early discussions which motivated the idea of FlyingTemplate.

10. REFERENCES

[1] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the General Track: 2000 USENIX Annual Technical Conference, June 18-23, 2000, San Diego, CA, USA*, pages 307–322. USENIX, 2000.

[2] J. Arnoldus, J. Bijpost, and M. van den Brand. Repleo: a syntax-safe template engine. In *Proceedings of GPCE 2007, 6th International Conference, Generative Programming and Component Engineering, Salzburg, Austria, October 1-3, 2007*, pages 25–32. ACM, 2007.

[3] Ashley IT Services Inc. RSLite demo. http://www.ashleyit.com/rs/rslite/.

[4] H. Böttger, A. Møller, and M. I. Schwartzbach. Contracts for cooperation between Web service programmers and HTML designers. *Journal of Web Engineering*, 5(1):65–89, 2006.

[5] S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a modeling language for designing web sites. *Computer Networks: The International Journal of Computer and Telecommunication Networking*, 33(1-6):137–157, 2000.

[6] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *Proceedings of SOSP 2007, the 21st ACM Symposium on Operating Systems Principles 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 31–44. ACM, 2007.

[7] D. W. Cornell, D. M. Dias, and P. S. Yu. On multisystem coupling through function request shipping. *IEEE Transaction on Software Engineering*, 12(10):1006–1017, 1986.

[8] Document Object Model (DOM) level 3 core specification version 1.0. W3C Recommendation 07 April 2004, http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/.

[9] P. Dömel, A. Lingnau, and O. Drobnik. Mobile agent interaction in heterogeneous environments. In *Proceedings of MA'97, First International Workshop on Mobile Agents, Berlin, Germany, April 7-8, 1997*, LNCS 1219, pages 136–148. Springer, 1997.

[10] P. Fraternali. Tools and approaches for developing data-intensive web applications: A survey. *ACM Comput. Surv.*, 31(3):227–263, 1999.

[11] F. J. García, R. I. Castanedo, and A. A. J. Fuente. A double-model approach to achieve effective model-view separation in template based web applications. In *Proceedings of ICWE 2007, 7th International Conference on Web Engineering, Como, Italy, July 16-20, 2007*, LNCS 4607, pages 442–456. Springer, 2007.

[12] J. J. Garrett. Ajax: A new approach to web applications, February 2005. http://adaptivepath.com/ideas/essays/archives/000385.php.

[13] D. Goodman. *Dynamic HTML: The Definitive Reference*. O'Reilly, December 2006.

[14] A vocabulary and associated APIs for HTML and XHTML, 2008. W3C Working Draft 10 June 2008, http://www.w3.org/TR/2008/WD-html5-20080610/.

[15] Hypertext Transfer Protocol – HTTP/1.1, 1999. RFC 2616.

[16] The application/json media type for JavaScript object notation (JSON), July 19992006. RFC 4627.

[17] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *Proceedings of WWW 2008, the 17th International Conference on World Wide Web, Beijing, China, April 21-25, 2008*, pages 535–544. ACM, 2008.

[18] A. Leff and J. T. Rayfield. Web-application development using the model/view/controller design pattern. In *Proceedings of EDOC 2001, 5th International Enterprise Distributed Object Computing Conference, September 4-7, 2001, Seattle, WA, USA*, pages 118–127. IEEE Computer Society, 2001.

[19] New Digital Group, Inc. Smarty: Template engine. http://www.smarty.net/.

[20] T. J. Parr. Enforcing strict model-view separation in template engines. In *Proceedings of WWW 2004, the 13th international conference on World Wide Web, New York, NY, USA, May 17-20, 2004*, pages 224–233. ACM, 2004.

[21] PHP: Hypertext preprocessor. http://php.net/.

[22] Standard Performance Evaluation Corporation. SPECWeb2005, 2005. http://www.spec.org/web2005/.

[23] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" Java software. In L. Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, LNCS 2072, pages 236–255, Budapest, Hungary, June 2001. Springer-Verlag.

[24] E. Tilevich and Y. Smaragdakis. J-Ohrchestra: Automatic Java application partitioning. In B. Magnusson, editor, *ECOOP 2002 - Object Oriented Programming*, LNCS 2374, pages 178–204, Malaga, Spain, June 2002. Springer-Verlag.

[25] A. Tozawa, M. Tatsubori, T. Onodera, and Y. Minamide. Copy-on-write in the PHP language. In *Proceedings of POPL 2009, the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Savannah, Georgia, USA, January 21-23, 2009*, pages 200–212. ACM, 2009.

[26] S. Trent, M. Tatsubori, T. Suzumura, A. Tozawa, and T. Onodera. Performance comparison of PHP and JSP as server-side scripting languages. In *Proceedings of Middleware 2008, ACM/IFIP/USENIX 9th International Middleware Conference, Leuven, Belgium, December 1-5, 2008*, pages 164–182. Springer, 2008.

[27] The XMLHttpRequest object. W3C Working Draft 15 April 2008, http://www.w3.org/TR/2008/WD-XMLHttpRequest-20080415/.

[28] F. Yang, N. Gupta, N. Gerner, X. Qi, A. J. Demers, J. Gehrke, and J. Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *Proceedings of WWW 2007, the 16th International Conference on World Wide Web, Banff, Alberta, Canada, May 8-12, 2007*, pages 341–350. ACM, 2007.

[29] D. Yu, A. Chander, H. Inamura, and I. Serikov. Better abstractions for secure server-side scripting. In *Proceedings of WWW 2008, the 17th International Conference on World Wide Web, Beijing, China, April 21-25, 2008*, pages 507–516. ACM, 2008.