BRICS RS-01-17 Brabrand et al.: Language-Based Caching of Dynamically Generated HTML

BRICS

Basic Research in Computer Science

Language-Based Caching of Dynamically Generated HTML

Claus Brabrand
Anders Møller
Steffan Olesen
Michael I. Schwartzbach

BRICS Report Series RS-01-17

ISSN 0909-0878 May 2001

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

http://www.brics.dk
ftp://ftp.brics.dk
This document in subdirectory RS/01/17/

Language-Based Caching of Dynamically Generated HTML

Claus Brabrand, Anders Møller, Steffan Olesen, and Michael I. Schwartzbach
BRICS, Department of Computer Science
University of Aarhus, Denmark
{brabrand,amoeller,olesen,mis}@brics.dk

Abstract

Increasingly, HTML documents are dynamically generated by interactive Web services. To ensure that the client is presented with the newest versions of such documents it is customary to disable client caching causing a seemingly inevitable performance penalty. In the <bigwig> system, dynamic HTML documents are composed of higher-order templates that are plugged together to construct complete documents. We show how to exploit this feature to provide an automatic fine-grained caching of document templates, based on the service source code. A <bigwig> service transmits not the full HTML document but instead a compact JavaScript recipe for a client-side construction of the document based on a static collection of fragments that can be cached by the browser in the usual manner. We compare our approach with related techniques and demonstrate on a number of realistic benchmarks that the size of the transmitted data and the latency may be reduced significantly.

1 Introduction

One central aspect of the development of the World Wide Web during the last decade is the increasing use of *dynamically* generated documents, that is, HTML documents generated using e.g. CGI, ASP, or PHP by a server at the time of the request from a client [22, 2]. Originally, the hypertext documents on the Web were considered to be principally *static*, which has influenced the design of protocols and implementations. For instance, an important technique for saving bandwidth, time, and clock-cycles is to cache documents on the client-side. Using the original HTTP protocol, a document that never or rarely changes can be associated an "expiration time" telling the browsers and proxy servers that there should be no need to reload the document from the server before that time. However, for dynamically generated documents that change on every request, this feature must be disabled—the expiration time is always set to "now", voiding the benefits of caching.

Even though most caching schemes consider all dynamically generated documents "non-cachable" [20, 3], a few proposals for attacking the problem have emerged [24, 17, 8, 12, 7, 9]. However, as described below, these proposals are typically not applicable for highly dynamic documents. They are often based on the assumptions that although a document is dynamically generated, 1) its construction on the server often does not have side-effects, for instance because the request is essentially a database lookup operation, 2) it is likely that many clients provide the same arguments for the request, or 3) the dynamics is limited to e.g. rotating banner ads. We take the next step by considering complex services where essentially every single document shown to a client is unique and its construction has side-effects on the server. A typical example of such a service is a Web-board where current discussion threads are displayed according to the preferences of each user. What we propose is not a whole new caching scheme requiring intrusive modifications to the Web architecture, but rather a technique for exploiting the caches already existing on the client-side in browsers, resembling the suggestions for future work in [22].

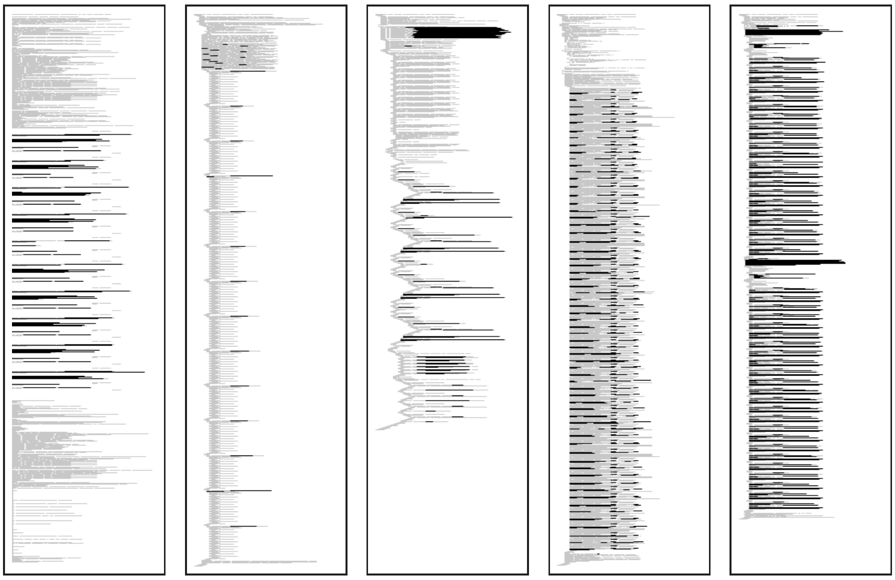
Though caching does not work for whole dynamically constructed HTML documents, most Web services construct HTML documents using some sort of constant templates that ought to be cached, as also observed in [9, 21]. In Figure 1, we show a condensed view of five typical HTML pages generated by different `<bigwig>` Web services [5] which are further described in Section 5. The grey sections, which appear to constitute a significant part, originate from a large number of small, constant HTML templates in the source code, while the black sections are dynamically built strings of character data. The main idea of this paper is—automatically, based on the source code of the Web service—to exploit this division into constant and dynamic parts in order to enable caching of the constant parts and provide an efficient transfer of the dynamic parts from the server to the client.

Using a technique based on JavaScript for shifting the actual HTML document construction from the server to the client, our contributions in this paper are:

- an automatic characterization, based on the source code, of document fragments as *cachable* or *dynamic*, permitting the standard browser caches to have significant effect even on dynamically generated documents;
- a *compact representation* of the information sent to the client for constructing the HTML documents; and
- a generalization allowing a whole group of documents, called a *document cluster*, to be sent to the client in a single interaction and cached efficiently.

All this is possible due to the unique approach for dynamically constructing HTML documents used in the `<bigwig>` language [18, 5], which we use as a foundation. Our technique builds only on preexisting technologies, such as HTTP and JavaScript—no special browser plug-ins, cache proxies, or server modules are employed, and no extra effort is required by the service programmer.

As a result, we obtain a simple and practically useful technique for saving network bandwidth and reviving the cache mechanism present in all modern Web browsers.



(a) lycos (b) bachelor (c) jaoo (d) webboard (e) dmodlog

Figure 1: Benchmark services: cachable (grey) vs. dynamic (black) parts.

Outline

Section 2 covers relevant related work. In Section 3, we describe the <bigwig> approach to dynamic generation of Web documents in a high-level language using HTML templates. Section 4 describes how the actual document construction is shifted from server-side to client-side. In Section 5, we evaluate our technique by experimenting with five <bigwig> Web services. Finally, Section 6 contains plans and ideas for further improvements.

2 Related Work

Caching of dynamic contents has received increasing attention the last years since it became evident that traditional caching techniques were becoming insufficient. In the following we present a brief survey of existing techniques that are related to the one we suggest.

Most existing techniques labeled "dynamic document caching" are either server-based, e.g. [17, 8, 12, 24], or proxy-based, e.g. [7, 19]. Ours is client-based, as e.g. the HPP language [9].

The primary goal for server-based caching techniques is not to lower the network load or end-to-end latency as we aim for, but to relieve the server by memoizing the generated documents in order to avoid redundant computations. Such techniques are orthogonal to the one we propose. The server-based techniques work well for services where many documents have been computed before, while our technique works well for services where every document is unique. Presumably, many services are a mixture of the two kinds, so these different approaches might support each other well—however, we do not examine that claim in this paper.

In [17], the service programmer specifies simple cache invalidation rules instructing a server caching module that the request of some dynamic document will make other cached responses stale. The approach in [24] is a variant of this with a more expressive invalidation rule language, allowing classes of documents to be specified based on arguments, cookies, client IP address, etc. The technique in [12] instead provides a complete API for adding and removing documents from the cache. That efficient but rather low-level approach is in [8] extended with *object dependency graphs*, representing data dependencies between dynamic documents and underlying data. This allows cached documents to be invalidated automatically whenever certain parts of some database are modified. These graphs also allow representation of *fragments* of documents to be represented, as our technique does, but caching is not on the client-side. A related approach for caching in the Weave Web site specification system is described in [23].

In [19], a protocol for proxy-based caching is described. It resembles many of the server-based techniques by exploiting equivalences between requests. A notion of *partial request equivalence* allows similar but non-identical documents to be identified, such that the client quickly can be given an approximate response while the real response is being generated.

Active Cache [7] is a powerful technique for pushing computation to proxies, away from the server and closer to the client. Each document can be associated a *cache applet*, a piece of code that can be executed by the proxy. This applet is able to determine whether the document is stale and if so, how to refresh it. A document can be refreshed either the traditional way by asking the server or, in the other extreme, completely by the proxy without involving the server, or by some combination. This allows tailor-made caching policies to be made, and—compared to the server-side approaches—it saves network bandwidth. The drawbacks of this approach are: 1) it requires installation of new proxy servers which can be a serious impediment to wide-spread practical use, and 2) since there is no general automatic mechanism for characterizing document fragments as cachable or dynamic, it requires tedious and error-prone programming of the cache applets whenever non-standard caching policies are desired.

Common to the techniques from the literature mentioned above is that truly dynamic documents, whose construction on the server often have side-effects and essentially always are unique (but contain common constant fragments), either cannot be cached at all or require a costly extra effort by the programmer for explicitly programming the cache. Furthermore, the techniques either are inherently server-based, and hence do not decrease network load, or require installation of proxy servers.

Delta encoding [15] is based on the observation that most dynamically constructed documents have many fragments in common with earlier versions. Instead of transferring the complete document, a *delta* is computed representing the changes compared to some common base. Using a cache proxy, the full document is regenerated near the client. Compared to Active Cache, this approach is automatic. The drawbacks are—in addition to requiring proxies—that it necessitates protocols for management of past versions, and it does not help with repetitions within a single document.

The HPP language [9] is closely related to our approach. Both are based on the observation that dynamically constructed documents usually contain common constant fragments. HPP is an HTML extension which allows an explicit separation between static and dynamic parts of a dynamically generated document. The static parts of a document are collected in a *template* file while the dynamic parameters are in a separate *binding* file. The template file can contain simple instructions, akin to embedded scripting languages such as ASP, PHP, or JSP, specifying how to assemble the complete document. According to [9], this assembly and the caching of the templates can be done either using cache proxies or in the browser with Java applets or plug-ins, but it should be possible to use JavaScript instead, as we do.

An essential difference between HPP and our approach is that the HPP solution is not integrated with the programming language used to make the Web service. With some work it should be possible to combine HPP with popular embedded scripting languages, but the effort of explicitly programming the document construction remains. Our approach is based on the source language, meaning that all caching specifications are automatically extracted from the Web service source code by the compiler and the programmer is not required to be aware of caching aspects. Regarding cachability, HPP has the advantage that the instructions describing the structure of the resulting document are located in the template file which is cached, while in our solution the equivalent information is in the dynamic file. However, in HPP the constant fragments constituting a document are collected in a single template. This means that HTML fragments that are common to different document templates cannot be reused by the cache. Our solution is more fine-grained since it caches the individual fragments separately. Also, HPP templates are highly specialized and hence more difficult to modify and reuse for the programmer. Being fully automatic, our approach guarantees cache soundness. Analogously to optimizing compilers, we claim that the `<bigwig>` compiler generates caching code that is competitive to what a human HPP programmer could achieve. This claim is substantiated by the experiments in Section 5. Moreover, we claim that `<bigwig>` provides a more flexible, safe, and hence easier to use template mechanism than does HPP or any other embedded scripting language. The `<bigwig>` notion of *type-safe higher-order templates as first-class values* is summarized in Section 3. A thorough comparison between various mechanisms supporting document templates can be found in [5].

As mentioned, we use compact JavaScript code to combine the cached and the dynamic fragments on the client-side. Alternatively, similar effects could be obtained using browser plug-ins or proxies, but implementation and installation would become more difficult. The HTTP 1.1 protocol [10] introduces both automatic compression using general-purpose algorithms, such as `gzip`, byte-range requests, and advanced cache-control directives. The compression features are essentially orthogonal to what we propose, as shown in Section 5. The byte-range and caching directives provide features reminiscent of our JavaScript code, but it would require special proxy servers or browser extensions to apply them to caching of dynamically constructed documents. Finally, we could have chosen Java instead of JavaScript, but JavaScript is more lightweight and is sufficient for our purposes.

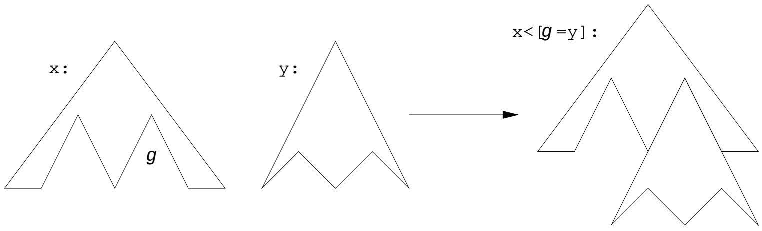


Figure 2: The *plug* operator.

3 Dynamic Documents in <bigwig>

The part of the <bigwig> Web service programming language that deals with dynamic construction of HTML documents is called DynDoc [18]. It is based on a notion of *templates* which are HTML fragments that may contain *gaps*. These gaps can at runtime be filled with other templates or text strings, yielding a highly flexible mechanism.

A <bigwig> *service* consists of a number of *sessions* which are essentially entry points with a sequential action that may be invoked by a client. When invoked, a session thread with its own local state is started for controlling the interactions with the client. Two built-in operations, *plug* and *show*, form the core of DynDoc. The *plug* operation is used for building documents. As illustrated in Figure 2, this operator takes two templates, `x` and `y`, and a gap name *g* and returns a copy of `x` where a copy of `y` has been inserted into every *g* gap. A template without gaps is considered a complete *document*. The *show* operation is used for interacting with the client, transmitting a given document to the client's browser. Execution of the client's session thread is suspended on the server until the client submits a reply. If the document contains input fields, the `show` statement must have a `receive` part for receiving the field values into program variables.

The templates are *first-class* values that can be passed and stored as any other data type. As in Mawl [13, 1], the programmer and designer tasks may then be completely separated. Since templates can be plugged into templates, the DynDoc template mechanism is *higher-order*, as opposed to Mawl where only strings can can be inserted into the gaps. The higher-order nature of our mechanism makes it more expressive, but without compromising runtime safety because of two compile-time program analyses: a *gap-and-field analysis* [18] and an *HTML validation analysis* [6]. The former analysis will guarantee that at every *plug*, the designated gap is actually present at runtime in the given template and at every *show*, there is always a valid correspondence between the input fields in the document being shown and the values being received. The latter analysis will guarantee that every document being shown is valid according to the HTML specification. The following variant of a well-known example illustrates the DynDoc concepts:

```
service {
  html ask = <html>What? <input name="what"></html>;
  html hello = <html>Hello, <b><[thing]></b>!</html>;
```

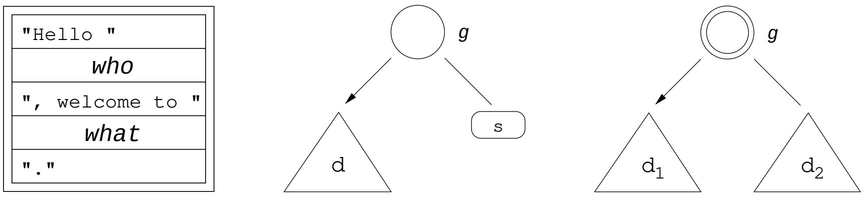
```
  session HelloWorld() {
    string s;
    show ask receive [s=what];
    hello = hello<[thing=s];
    show hello;
  }
}
```

Two HTML variables, `ask` and `hello`, are initialized with constant HTML templates, and a session `HelloWorld` is declared. The entities `<html>` and `</html>` are merely lexical delimiters and are not part of the actual templates. When invoked, the session first shows the `ask` template as a complete document to the client. All documents are implicitly wrapped into an `<html>` element and a form with a default "continue" button before being shown. The client fills out the *`what`* input field and submits a reply. The session resumes execution by storing the field value in the `s` variable. It then plugs that value into the *`thing`* gap of the `hello` template and sends the resulting document to the client. The following more elaborate example will be used throughout the paper:

```
service {
  html cover = <html>
    <head><title>Welcome</title></head>
    <body bgcolor=[color]>
      <[contents]>
    </body>
  </html>;

  html greeting = <html>
    Hello <[who]>, welcome to <[what]>.
  </html>;

  html person = <html><i>Stranger</i></html>;

  session welcome() {
    html h;
    h = cover<[color="#9966ff",
               contents=greeting<[who=person]];
    show h<[what=<html><b>BRICS</b></html>];
  }
}
```

It builds a "welcome to BRICS" document by plugging together four constant templates and a single text string, shows it to the client, and terminates.

To illustrate that our higher-order templates are more expressive and provide better cachability compared to first-order template mechanisms, consider for instance the unbounded hierarchical list of messages in a typical Web bulletin board. This is easily expressed recursively using a small collection of DynDoc templates. However, it can never be captured by any first-order solution without casting from templates to strings



(a) Leaf: `greeting` (b) Node: `strplug(d,g,s)` (c) Node: `plug(d`$_1$`,g,d`$_2$`)`

Figure 4: DynDocDag representation constituents.

and hence losing type safety and cachability potential. Figure 3 shows the `webboard` benchmark as it would appear if it had been generated using first-order templates only. Nearly everything is dynamic (black) compared to the higher-order version displayed in Figure 1(d).

All in all, we have with the *plug-and-show* mechanism in `<bigwig>` successfully transferred many of the advantages known from static documents to a dynamic context. The next step, of course, being caching.

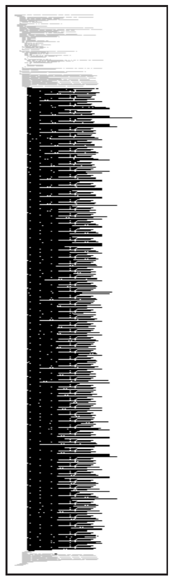


Figure 3: `webboard`

Dynamic Document Representation

Dynamic documents in `<bigwig>` are at runtime represented by the DynDocDag data structure [4] supporting four operations: constructing constant templates, `constant(c)`; string plugging, `strplug(d,g,s)`; template plugging, `plug(d`$_1$`,`g`,d`$_2$`)`; and showing documents, `show(d)`. This data structure represents a dynamic document as a binary DAG (Directed Acyclic Graph), where the leaves are either HTML templates or strings that have been plugged into the document and where the nodes represent pluggings that have constructed the document.

A constant template is represented as an ordered sequence of its text and gap constituents. For instance, the `greeting` template from the BRICS example service is represented as displayed in Figure 4(a) as a sequence containing two gap entries, *who* and *what*, and three text entries for the text around and between the gaps. A constant template is represented only *once* in memory and is shared among the documents it has been plugged into, causing the data structure to be a DAG in general and not a tree.

The string plug operation, `strplug`, combines a DAG and a constant string by adding a new string plug root node with the name of the gap, as illustrated in Figure 4(b). Analogously, the `plug` operation combines two DAGs as shown in Figure 4(c). For both operations, the left branch is the document containing the gap being

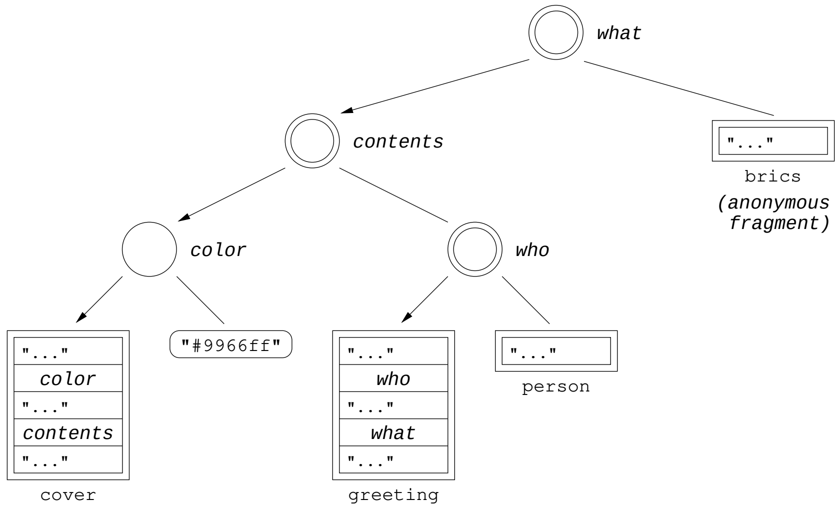


Figure 5: DynDocDag representation of the document shown in the BRICS example.

plugged and the right branch is the value being plugged into the gap. Thus, the data structure merely records plug operations and defers the actual document construction to subsequent `show` operations.

Conceptually, the `show` operation is comprised of two phases: a *gap linking* phase that will insert a stack of links from gaps to templates and a *print traversal* phase that performs the actual printing by traversing all the gap links. The need for stacks comes from the template sharing.

The `strplug(d,`*g*`,s)`, `plug(d`$_1$`,`*g*`,d`$_2$`)`, and `show(d)` operations have optimal complexities, $O(1)$, $O(1)$, and $O(|\mathtt{d}|)$, respectively, where $|\mathtt{d}|$ is the lexical size of the `d` document. We refer to [4] (and the earlier paper [18]) for a more formal specification and complexity analysis of the DynDoc unfolding algorithm.

Figure 5 shows the representation of the document shown in the BRICS example service. In this simple example, the DAG is a tree since each constant template is used only once. Note that for some documents, the representation is exponentially more succinct than the expanded document. This is for instance the case with the following recursive function:

```
html tree(int n) {
  html list = <html><ul><li><[gap]><li><[gap]></ul></html>;
  if (n==0) return <html>foo</html>;
  return list<[gap=tree(n-1)];
}
```

which, given n, in $O(n)$ time and space will produce a document of lexical size $O(2^n)$. This shows that regarding network load, it can be highly beneficial to transmit the DAG across the network instead of the resulting document, even if ignoring cache aspects.

4 Client-Side Caching

In this section we will show how to cache reoccurring parts of dynamically generated HTML documents and how to store the documents in a compact representation. The first step in this direction is to move the unfolding of the DynDocDag data structure from the server to the client. Instead of transmitting the unfolded HTML document, the server will now transmit a DynDocDag representation of the document in JavaScript along with a link to a file containing some generic JavaScript code that will interpret the representation and unfold the document on the client. Caching is then obtained by placing the constant templates in separate files that can be cached by the browser as any other files.

As we shall see in Section 5, both the caching and the compact representation substantially reduce the number of bytes transmitted from the server to the client. The compromise is of course the use of client clock cycles for the unfolding, but in a context of fast client machines and comparatively slow networks this is a sensible tradeoff. As explained earlier, the client-side unfolding is not a computationally expensive task, so the clients should not be too strained from this extra work, even with an interpreted language like JavaScript.

One drawback of our approach is that extra TCP connections are required for downloading the template files the first time, unless using the "keep connection alive" feature in HTTP 1.1. However, this is no worse than downloading a document with many images. Our experiments show that the number of transmissions per interaction is limited, so this does not appear to be a practical problem.

4.1 Caching

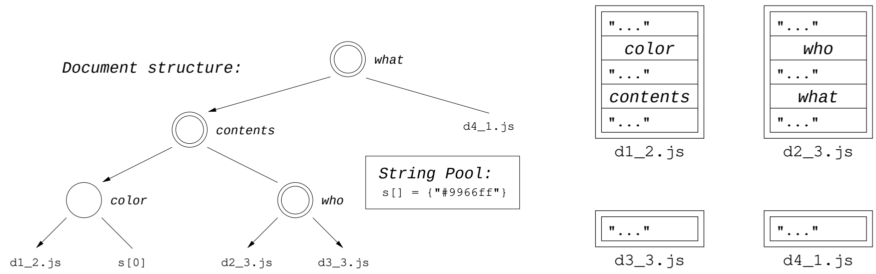
The DynDocDag representation has a useful property: it explicitly maintains a separation of the *constant templates* occurring in a document, the *strings* that are plugged into the document, and the *structure* describing how to assemble the document. In Figure 5, these constituents are depicted as framed rectangles, oval rectangles, and circles, respectively.

Experiments suggest that templates tend to occur again and again in documents shown to a client across the lifetime of a `<bigwig>` service, either because they occur 1) many times in the same document, 2) in many different documents, or 3) simply in documents that are shown many times. The strings and the structure parts, however, are typically dynamically generated and thus change with each document.

The templates account for a large portion of the expanded documents. This is substantiated by Figure 1, as earlier explained. Consequently, it would be useful to somehow cache the templates in the browser and to transmit only the dynamic parts, namely the strings and the structure at each `show` statement. This separation of cachable and dynamic parts is for the BRICS example illustrated in Figure 6.

As already mentioned, the solution is to place each template in its own file and include a link to it in the document sent to the client. This way, the caching mechanism in the browser will ensure that templates already seen are not retransmitted.

The first time a service shows a document to a client, the browser will obviously not have cached any of the JavaScript template files, but as more and more documents



(a) Dynamic document structure reply file.

(b) Cachable template files.

Figure 6: Separation into cachable and dynamic parts.

are shown, the client will download fewer and fewer of these files. With enough interactions, the client reaches a point of *asymptotic caching* where all constant templates have been cached and thus only the dynamic parts are downloaded.

Since the templates are statically known at compile-time, the compiler enumerates the templates and for each of them generates a file containing the corresponding JavaScript code. By postfixing template numbers with version numbers, caching can be enabled across recompilations where only some templates have been modified.

4.2 Compact Representation

In the following we show how to encode the cachable template files and the reply documents containing the document representation. Since the reply documents are transmitted at each `show` statement, their sizes should be small. Decompression has to be conducted by JavaScript interpreted in browsers, so we do not apply general purpose compression techniques. Instead we exploit the inherent structure of the reply documents to obtain a lightweight solution: a simple yet compact JavaScript representation of the string and structure parts that can be encoded and decoded efficiently.

Constant Templates

A constant template is placed in its own file for caching and is encoded as a call to a JavaScript constructor function, `F`, that takes the number and version of the template followed by an array of text and gap constituents respectively constructed via calls to the JavaScript constructor functions `T` and `G`. For instance, the `greeting` template from the BRICS example gets encoded as follows:

```
F(T('Hello '),G(3),T(', welcome to '),G(4),T('.'));
```

Assuming this is version 3 of template number 2, it is placed in a file called `d2_3.js`. The gap identifiers *who* and *what* have been replaced by the numbers 3 and 4, respectively, abstracting away the identifier names. Note that such a file needs only ever be downloaded once by a given client, and it can be reused every time this template occurs in a document.

Dynamics

The JavaScript reply files transmitted at each `show` contain three document specific parts: *include directives* for loading the cachable JavaScript template files, the *dynamic structure* showing how to assemble the document, and a *string pool* containing the strings used in the document.

The structure part of the representation is encoded as a JavaScript string constant, using a uuencode-like scheme which is tuned to the kinds of DAGs that occur in the observed benchmarks.

Empirical analyses have exposed three interesting characteristics of the strings used in a document: 1) they are all relatively short, 2) some occur many times, and 3) many seem to be URLs and have common prefixes. Since the strings are quite short, placing them in individual files to be cached would drown in transmission overhead. For reasons of security, we do not want to bundle up all the strings in cachable string pool files. This along with the multiple occurrences suggests that we collect the strings from a given document in a string pool which is inlined in the reply file sent to the client. String occurrences within the document are thus designated by their offsets into this pool. Finally, the common prefix sharing suggests that we collect all strings in a *trie* which precisely yields sharing of common prefixes. As an example, the following four strings:

```
"foo",
"http://www.brics.dk/bigwig/",
"http://www.brics.dk/bigwig/misc/gifs/bg.gif",
"http://www.brics.dk/bigwig/misc/gifs/bigwig.gif"
```

are linearized and represented as follows:

```
"foo|http://www.brics.dk/bigwig/[misc/gifs/b(igwig.gif|g.gif)]"
```

The reply document transmitted to the client at the `show` statement in the BRICS example looks like:

```
<html>
 <head>
  <script src="http://www.brics.dk/bigwig/dyndoc.js"></script>
  <script>I(1,2,3,4, 2,3,3,1);</script>
  <script>S("#9966ff");  D("/&Ë$Î&I%",2,8,4);</script>
 </head>
 <body onload="E();"></body>
</html>
```

The document starts by including the generic JavaScript library, `dyndoc.js`, for unfolding the DynDocDag representation. This file is shared among all services and will thus get cached after the first service interaction. The include directives are encoded as calls to the function I whose argument is an array designating the template files that are to be included in the document along with their version numbers. The S constructor function reconstructs the string trie which in our example contains the only string plugged into the document, namely "#9966ff". As expected, the document structure part, which is reconstructed by the D constructor function, is not humanly readable. The last three arguments to D recount how many bytes are used in the encoding of a node, the number of templates plus plug nodes, and the number of gaps, respectively. The last line of the document calls the JavaScript function E that will interpret all constituents to expand the document. After this, the document has been fully replaced by the expansion. Note that three script sections are required to ensure that processing occurs in distinct phases and dependencies are resolved correctly. Viewing the HTML source in the browser will display the resulting HTML document, not our encodings.

Our compact representation makes no attempts at actual compression such as gzip or XML compression [14], but is highly efficient to encode on the server and to decode in JavaScript on the client. Compression is essentially orthogonal in the sense that our representation works independently of whether or not the transmission protocol compresses documents sent across the network, as shown in Section 5. However, the benefit factor of our scheme is of course reduced when compression is added.

4.3 Clustering

In <bigwig>, the show operation is not restricted to transmit a single document. It can be a collection of interconnected documents, called a *cluster*. For instance, a document with input fields can be combined in a cluster with a separate document with help information about the fields.

A hypertext reference to another document in the same cluster may be created using the notation &x to refer to the document held in the HTML variable x at the time the cluster is shown. When showing a document containing such references, the client can browse through the individual documents without involving the service code. The control-flow in the service code becomes more clear since the interconnections can be set up as if the cluster were a single document and the references were internal links within it.

The following example shows how to set up a cluster of two documents, input and help, that are cyclically connected with input being the main document:

```
service {
  html input = <html>
    Please enter your name: <input name="name"><p>
    Click <a href=[help]>here</a> for help.
  </html>;

  html help = <html>
    You can enter your given name, family name, or nickname.
    <p><a href=[back]>Back</a> to the form.
  </html>;

  html output = <html>Hello <[name]>!</html>;
```

```
  session cluster_example() {
    html h, i;
    string s;
    h = help<[back=&i];
    i = input<[help=&h];
    show i receive [s=name];
    show output<[name=s];
  }
}
```

The cluster mechanism gives us a unique opportunity for further reducing network traffic. We can encode the entire cluster as a single JavaScript document, containing all the documents of the cluster along with their interconnections. Wherever there is a document reference in the original cluster, we generate JavaScript code to overwrite the current document in the browser with the referenced document of the cluster. Of course, we also need to add some code to save and restore entered form data when the client leaves and re-enters pages with forms. In this way, everything takes place in the client's browser and the server is not involved until the client leaves the cluster.

5 Experiments

Figure 7 recounts the experiments we have performed. We have applied our caching technique to five Web services that each generate a complex dynamic HTML page. The `lycos` example is inspired by HPP [9] and simulates a search engine giving 10 results from the query "caching dynamic objects"; the `bachelor` service generates based on a course roster a list of menus that students use to plan their studies; the `jaoo` service is part of a conference administration system and generates a graphical schedule of events; the `webboard` service generates a hierarchical list of active discussion threads; and the `dmodlog` service generates lists of participants in a course. Apart from the first simulation, all these examples are sampled from running services and use real data. The `dmodlog` example is dominated by dynamic string data, as seen in Figure 1, as is thus included as a worst-case scenario for our technique.

In Figure 7(a) we show the sizes of the data transmitted to the client. The grey columns show the original document sizes, ranging between 20 and 90 KB. The white columns show the sizes of the total data that is transmitted using our technique, none of which exceeds 20 KB. Of ultimate interest is the black column which shows the asymptotic sizes of the transmitted data, when the templates have been cached by the client. In this case, we see reductions of factors between 4 and 37 compared to the original document size.

The `lycos` benchmark is similar to one presented for HPP [9], except that our reconstruction is of course in `<bigwig>`. It is seen that the size of our residual dynamic data (from 20,183 to 3,344 bytes) is virtually identical to that obtained by HPP (from 18,000 to 3,250 bytes). However, that solution is hand-coded with the benefit of human insight, while ours is automatically generated by the `<bigwig>` compiler. The other four benchmarks would be more challenging for HPP.

In Figure 7(b) we repeat the comparisons from Figure 7(a) but under the assumption that the data is transmitted compressed using `gzip`. Of course, this drastically reduces the benefits of our caching technique. However, we still see asymptotic reduction factors between 1.3 and 2.9 suggesting that our approach remains worthwhile even in these circumstances. Clearly, there are documents for which the asymptotic reduction factors will be arbitrarily large, since large constant text fragments count for zero on our side of the scales while `gzip` can only compress them to a certain size. Hence we feel justified in claiming that compression is orthogonal to our approach. When the HTTP protocol supports compression, we represent the string pool in a naive fashion rather than as a trie, since `gzip` does a better job on plain data. Note that in some cases our uncompressed residual dynamic data is smaller that the compressed version of the original document.

In Figure 7(c) we quantify the end-to-end latency for our technique. The total download and rendering times for the five services are shown for both the standard documents and our cached versions. The client is Internet Explorer 5 running on an 800 MHz Pentium III Windows PC connected to the server by a 28.8K modem. This is still a realistic configuration, since by August 2000 the vast majority of Internet subscribers used dial-up connections [11] and this situation will not change significantly within the next couple of years [16]. The times are averaged over several downloads plus renderings. As expected, this yields dramatic reduction factors between 2.1 and 9.7. For higher bandwidth dimensions, the results will of course be less impressive.

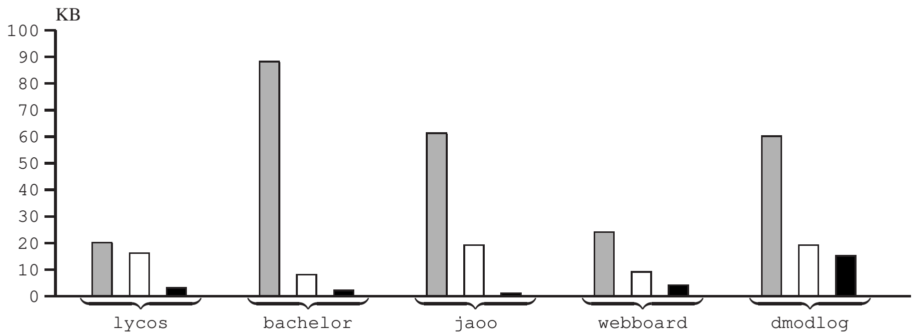
In Figure 7(d) we focus on the pure rendering times, ignoring the download latencies. For the first three benchmarks, our times are a bit faster than for the original HTML documents. For the last two benchmarks, they are somewhat slower. These figures are of course dependent on the quality of the JavaScript interpreter that is available in the browser.

6 Future Work

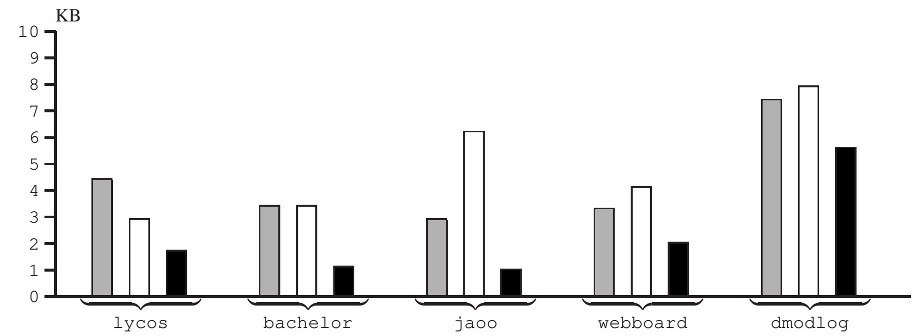
In the following, we describe a few ideas for further cutting down the number of bytes and files transmitted between the server and the client.

In many services, certain templates often occur together in all `show` statements. Such templates could be grouped in the same file for caching, thereby lowering the transmission overhead. In `<bigwig>`, the HTML validation analysis [6] already approximates a graph from which we can readily derive the set of templates that can reach a given `show` statement. These sets could then be analyzed for tightly connected templates using various heuristics. However, there are certain security concerns that need to be taken into consideration. It might not be good idea to indirectly disclose a template in a cache bundle if the show statement does not directly include it.

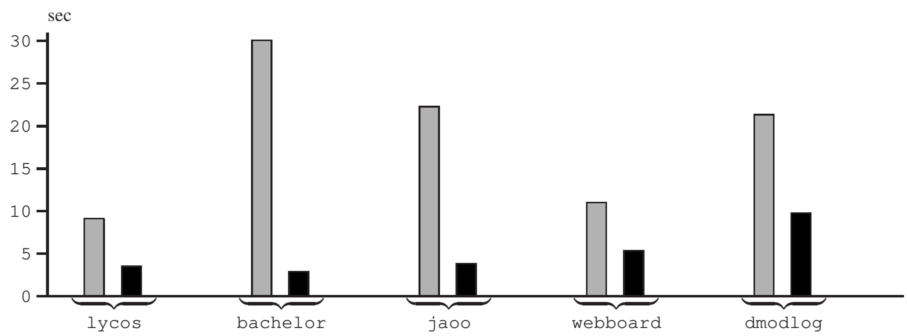
Finally, it is possible to also introduce language-based server-side caching, where we exploit the structure of `<bigwig>` programs to automatically cache and invalidate the documents being generated, resembling the server-side caching techniques mentioned in Section 2.



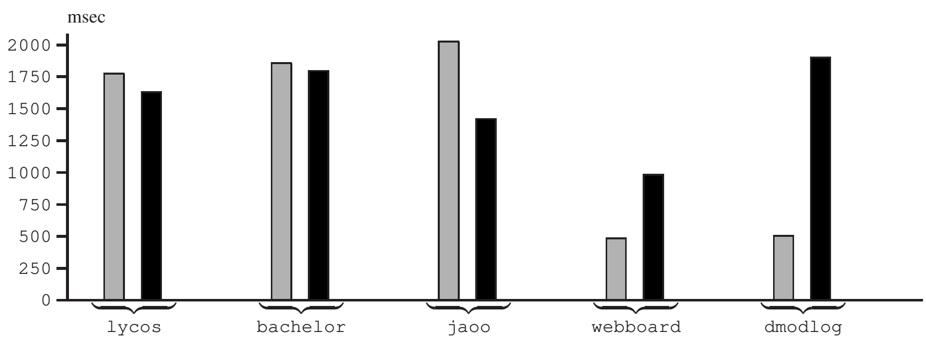
(a) size (original, cachable+dynamics, dynamics)



(b) `gzip` size (original, cachable+dynamics, dynamics)



(c) download+rendering (original, dynamics)



(d) pure rendering (original, dynamics)

Figure 7: Experiments with the template representation.

7 Conclusion

We have presented a technique to revive the existing client-side caching mechanisms in the context of dynamically generated Web pages. With our approach, the programmer need not be aware of caching issues since the decomposition of pages into cachable and dynamic parts is performed automatically by the compiler. The resulting caching policy is guaranteed to be sound, and experiments show that it results in significantly smaller transmissions and reduced latency. Our technique requires no extensions to existing protocols, clients, servers, or proxies. We only exploit that the browser can interpret JavaScript code. These results lend further support to the unique design of dynamic documents in `<bigwig>`.

References

[1] David Atkins, Thomas Ball, Glenn Bruns, and Kenneth Cox. Mawl: a domain-specific language for form-based services. In *IEEE Transactions on Software Engineering*, June 1999.

[2] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in web client access patterns: Characteristics and caching implications. Technical Report BUCS-1998-023, CS Dept., Boston University, November 1998.

[3] Greg Barish and Katia Obraczka. World Wide Web caching: Trends and techniques. *IEEE Communications Magazine Internet Technology Series*, May 2000.

[4] Claus Brabrand, Anders Møller, Steffan Olesen, and Michael I. Schwartzbach. Flexible and efficient dynamic documents. In preparation, 2001.

[5] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. Submitted for publication. Available from `http://www.brics.dk/bigwig/`.

[6] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001)*. ACM, 2001.

[7] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the Web. In *Proceedings of the 1998 Middleware conference*, 1998.

[8] Jim Challenger, Paul Dantzig, and Arun Iyengar. A scalable system for consistently caching dynamic web data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, March 1999.

[9] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, December 1997.

[10] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol, HTTP/1.1. Available from `http://www.w3.org/Protocols/rfc2616/rfc2616.html`, 1999.

[11] ICONOCAST Newsletter, August 17, 2000. Available from `http://www.iconocast.com/issue/20000817.html`.

[12] Arun Iyengar and Jim Challenger. Improving web server performance by caching dynamic data. In *USENIX Symposium on Internet Technologies and Systems*, December 1997.

[13] David A. Ladd and J. Christopher Ramming. Programming the web: An application-oriented language for hypermedia services. In *4th Intl. World Wide Web Conference (WWW4)*, 1995.

[14] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. Technical Report MS-CIS-99-26, AT&T, 1999. Available from `http://www.research.att.com/sw/tools/xmill/`.

[15] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *SIGCOMM*, pages 181–194, 1997.

[16] Jakob Nielsen. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, 2000.

[17] Karthick Rajamani and Alan Cox. A simple and effective caching scheme for dynamic content. Technical report, CS Dept., Rice University, September 2000.

[18] Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic Web documents. In *Principles of Programming Languages (POPL'00)*. ACM, 2000.

[19] Ben Smith, Anurag Acharya, Tao Yang, and Huican Zhu. Exploiting result equivalence in caching dynamic web content. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[20] Jia Wang. A survey of web caching schemes for the Internet. *ACM Computer Communication Review*, 29(5):36–46, October 1999.

[21] Craig Wills and Mikhail Mikhailov. Studying the impact of more complete server information on web caching. February 2000.

[22] Alec Wolman. Characterizing web workloads to improve performance, July 1999. University of Washington.

[23] Khaled Yagoub, Daniela Florescu, Valerie Issarny, and Patrick Valduriez. Caching strategies for data-intensive web sites. *The VLDB Journal*, pages 188–199, 2000.

[24] H. Zhu and T. Yang. Class-based cache management for dynamic web contents. Technical Report TRCS00-13, Dept. of Computer Science, University of California, Santa Barbara, 2000.

Recent BRICS Report Series Publications

RS-01-17 **Claus Brabrand, Anders Møller, Steffan Olesen, and Michael I. Schwartzbach.** ***Language-Based Caching of Dynamically Generated HTML.*** **May 2001. 18 pp.**

RS-01-16 **Olivier Danvy, Morten Rhiger, and Kristoffer H. Rose.** ***Normalization by Evaluation with Typed Abstract Syntax.*** **May 2001. 9 pp. To appear in *Journal of Functional Programming*.**

RS-01-15 **Luigi Santocanale.** ***A Calculus of Circular Proofs and its Categorical Semantics.*** **May 2001. 30 pp.**

RS-01-14 **Ulrich Kohlenbach and Paulo B. Oliva.** ***Effective Bounds on Strong Unicity in L_1-Approximation.*** **May 2001.**

RS-01-13 **Federico Crazzolara and Glynn Winskel.** ***Events in Security Protocols.*** **April 2001.**

RS-01-12 **Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær.** ***The Abstraction and Instantiation of String-Matching Programs.*** **April 2001.**

RS-01-11 **Alexandre David and M. Oliver Möller.** ***From* HUPPAAL *to* UPPAAL*: A Translation from Hierarchical Timed Automata to Flat Timed Automata.*** **March 2001. 40 pp.**

RS-01-10 **Daniel Fridlender and Mia Indrika.** ***Do we Need Dependent Types?*** **March 2001. 6 pp. Appears in *Journal of Functional Programming*, 10(4):409–415, 2000. Superseeds BRICS Report RS-98-38.**

RS-01-9 **Claus Brabrand, Anders Møller, and Michael I. Schwartzbach.** ***Static Validation of Dynamically Generated HTML.*** **February 2001. 18 pp.**

RS-01-8 **Ulrik Frendrup and Jesper Nyholm Jensen.** ***Checking for Open Bisimilarity in the π-Calculus.*** **February 2001. 61 pp.**

RS-01-7 **Gregory Gutin, Khee Meng Koh, Eng Guan Tay, and Anders Yeo.** ***On the Number of Quasi-Kernels in Digraphs.*** **January 2001. 11 pp.**

RS-01-6 **Gregory Gutin, Anders Yeo, and Alexey Zverovich.** ***Traveling Salesman Should not be Greedy: Domination Analysis of Greedy-Type Heuristics for the TSP.*** **January 2001. 7 pp.**