HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching

Fred Douglis*
AT&T Labs – Research

Antonio Haro†
College of Computing, Georgia Institute of Technology

Michael Rabinovich‡
AT&T Labs – Research

Abstract

A number of techniques are available for reducing latency and bandwidth requirements for resources on the World Wide Web, including caching, compression, and delta-encoding [12]. These approaches are limited: much data on the Web is dynamic, for which traditional caching is of limited use, and delta-encoding requires both a common version base against which to apply a delta and the complete generation of the resource prior to encoding it. In contrast to these approaches, we take an application-specific view, in which we separate the static and dynamic portions of a resource. The static portions (called the *template*) can then be cached, with (presumably small) dynamic portions obtained on each access. Our HTML extension, which we refer to as HPP (for **H**TML **P**re-**P**rocessing) supports resources that contain variable number of static and dynamic elements, such as query responses.

Results with macro-encoding of query response resources from local CGI scripts and two popular search engines indicate that our approach promises a substantial reduction of network traffic, server load, and access latency for dynamic documents. The size of network transfers using HPP are comparable to delta-encoding (factors of 2–8 smaller than the original resource), while the data generated by content providers is simpler, and the load on the end-servers is slightly lower.

*Email: douglis@research.att.com.

†This work was done while the author was visiting AT&T Labs–Research. Email: haro@cc.gatech.edu.

‡Email: misha@research.att.com.

1 Introduction

Caching plays a crucial role in a wide-area distributed system such as the World Wide Web. It significantly reduces response time for accessing cached resources by eliminating long-haul transmission delays. It also reduces backbone traffic and the load on content-providers.

The importance of caching will increase dramatically once ISDN lines and cable modems replace slow modems as the "last link" to the user. Indeed, a slow link to the user currently serves as a "floodgate" that limits the rate of requests from this user. With an order-of-magnitude increase in bandwidth provided by ISDN and cable modems, these floodgates will open, shifting the bottleneck further to the backbone and content servers.

However, a significant portion of Web resources are not cacheable, either because the resource is modified upon every access, or because the content provider explicitly prohibits caching [4]. Thus, increasingly sophisticated caching techniques are applied to a decreasing portion of resources on the Web.

Some proposals have been made to transmit encodings of the changes between subsequent versions of a resource, in order to reduce bandwidth requirements and improve end-to-end performance [1, 8, 12, 14]. With delta-encoding, one might cache a resource even if it is considered uncacheable, but not present the cached data without first obtaining the changes to present its current version. One advantage of these proposals is that they apply uniformly to all uncacheable resources regardless of the reason why they are uncacheable. Another advantage is that they can be implemented transparently, via proxies, so that content providers need not be modified. However, the delta-encoding proposals have disadvantages as well. If the content provider must compute the delta-encodings on the fly, it suffers overhead and must store a potentially large number of past versions; if the delta computation is performed by an intermediary, then the entire resource must be sent from the content provider to the intermediary, and the encoding must still be performed on the "critical path."

We have observed that with a common class of resources, such as those provided by search engines, a significant part of the resource is essentially static. Portions of the resource vary to different extents from one response to another (the difference between two pages in response to a single query is usually smaller than the difference between pages from different queries). Also, the location of the dynamic portions relative to the rest of the resource does not change. Consider, for example, a document generated in response to a stock quote query. It contains the banner identifying the content provider, headers, information specifying formatting and fonts, and, finally, the name and the stock price of the requested company. The banner, headers, and formatting stay the same, and the dynamic portion (the name and the stock price) go in the same place within the resource.

We therefore have extended HTML to allow the explicit separation of static and dynamic portions of a resource. The static portion contains macro-instructions for inserting dynamic information. The static portion together with these instructions (the *template*) can be cached freely. The dynamic portion contains the bindings of macro-variables to strings specific to the given access. The bindings are obtained for every access, and the template is expanded by the client prior to rendering the document. In other words, a macro-preprocessing phase at the client permits *partial* caching of dynamic resources.

We designed our macro-encoding language, which we refer to as HPP (for **H**TML **P**re-**P**rocessing), to minimize the size of the bindings. We motivate our proposal by examples from several popular resources, all of which show a remarkable difference in size between the original resource and the bindings: factors of 4–8 without compression, or 2–4 when comparing compressed bindings to the compressed original resource.

In addition to gains in performance, HPP should make authoring dynamic resources easier. Instead of writing programs that generate the full HTML document, the bulk of the document can be generated using an HTML authoring tool similar to the ones available now, and only the dynamic portions will have to be produced as a program output. In fact, similar techniques are already used to simplify programming (the "shtml" server-side include feature of many HTTP servers). Our proposal allows a systematic and more general way of doing this, and also exploits this separation to enable caching. Macro-preprocessing is almost a client-side equivalent to server-side inclusion, but with a slightly more sophisticated language that supports, for instance, looping constructs.

The rest of the paper is organized as follows. We discuss the extension to HTML for client-side macro-preprocessing in Section 2. Section 3 outlines the implementation path of our approach within the HTTP protocol and HTML language. Section 4 evaluates the performance of our approach using several existing dynamic resources including two well-known search engines. A more detailed comparison with related work is given in Section 5. We conclude in Section 6 with a summary of main results.

2 HTML Extension

Our proposal for macro-preprocessing within HTML is similar to source code preprocessing (e.g., *cpp*). In our case, the template is first expanded into the actual HTML document, which is then rendered on the browser screen. The HTML extension contains the following new tags: VAR, LOOP, IF, SET_VAR, and DYNAMICS. VAR, LOOP, IF, and SET_VAR are used in the template to compose instructions regarding the location of dynamic content. DYNAMICS is used to delineate the dynamic part of the document. It contains bindings of the macro-variables to dynamic text strings specific to the current document access.

2.1 Overview

The syntax of the new tags conforms with the SGML specification [6]. We describe these tags informally using a series of examples, without spending time on straightforward details.

The VAR tag is used to include a macro-expression in the text. The operations defined in the macro-expression are concatenation, basic arithmetic operations (which obviously make sense only if the operands evaluate to numbers), etc.

In many cases, a macro-expression contains a single variable and is used in the template as a placeholder, to be replaced by a text segment dynamically bound to this variable. An example of this approach appears in Figure 1 with "time" and "count" variables.

As a more complicated example, consider the response from Lycos[1] [11] to the query "caching dynamic objects." At the end of the pageful of results, the response

```
<HEAD>
<TITLE>Michael Rabinovich</TITLE>
<BODY>
...
The time is <VAR time>.
This page has been accessed <VAR count> times.
</BODY>
```

(a) The template for a simple query.

```
<HTML>
<TEMPLATE HREF="query.hpp">
<DYNAMICS>
time = 1:15pm;
count = 10;
</DYNAMICS>
</HTML>
```

(b) The bindings for a particular instantiation.

Figure 1: Example of HTML preprocessing encoding scheme.

```
<FONT SIZE=+1>1</FONT>
 . 
<A HREF="/cgi-bin/pursuit?first=11&part=&cat=lycos&query=caching+dynamic+objects">2</A>
 . 
<A HREF="/cgi-bin/pursuit?first=21&part=&cat=lycos&query=caching+dynamic+objects">3</A>
<...>
```

Figure 2: Output from the sample Lycos query, with links to additional pages of results.

provides links to the next ten pages of the results. These links are generated by the HTML fragment in Figure 2. This fragment is rendered in the browser window as 1.2.3.4.5.6.7.8.9.10, where each number is associated with a query URL. The URL contains the keywords and the number of the first result of interest. This fragment could be encoded in our extended HTML as the template and bindings shown in Figure 3, which illustrates the use of a more complex VAR expression, as well as the simple use of a LOOP tag. All variables in the DYNAMICS portion of the resource can be specified explicitly, separated by commas, or as a numeric range similar to a FOR loop with range and increment.

To arrive at the actual fragment, the pre-processor would expand the template fragment within the loop as many times as the number of bindings to the subcount macro-variable specified in the loop fragment of the DYNAMICS section. Each time the expansion is done with the new binding for the subcount variable. However, the same binding for the query variable is used in each expansion because the binding is specified outside the LOOP. Notice that the amount of dynamic information that changes with each access is significantly smaller in our encoding.

The next example concerns conditional macro-expansion. Consider, again, the fragment of Lycos output showing the numbers of ten pages of results. The number of the currently viewed page is included as plain text, while other page numbers are part of the anchors referring to corresponding URLs as shown in Figure 2. For example, if the current page is 4, the fragment would be rendered as 1.2.3.4.5.6.7.8.9.10. This fragment could be macro-encoded by splitting the loop of Figure 3 into two, with the first loop generating page numbers preceding the current page, followed by the current page number in a VAR expression, followed by the second loop generating the remaining page numbers. Another complication is that there are ten numbers in the fragment but only nine dots in-between. So, without extra functionality, one would have to encode the first number separately outside the loops, since each loop iteration adds a dot and a number. Figure 4 illustrates a more convenient way of macro-encoding the same fragment using conditional statements and assignments to macro-variables. We have not fully implemented conditional statements and assignments yet. Consequently, we hard-coded the first page number in the template as in Figure 2, assuming that separate templates are prepared for the second, third, etc pagefuls of results. Note that our shortcut caused negligible decrease in the size of the template and bindings compared to the encoding with conditionals and assignments (0.85% decrease for the template and 0.35% decrease for the bindings), so it does not affect our performance conclusions.

Our final example illustrates a more complex use of a LOOP construct. Considering the same query to Lycos as before, Figure 5 shows the fragment of the response that generates a list of ten results (only the first and the

```
<FONT SIZE=+1>1</FONT>
<loop>  . 
<A HREF="/cgi-bin/pursuit?first=
<var 10*(subcount - 1)+1>&part=&cat=lycos&
query=<var +query>"><var subcount></A>
</loop>
```

(a) The template.

```
query = caching+dynamic+objects
<loop>
subcount = 2 to 10,1;
</loop>
```

(b) The bindings.

Figure 3: Example of HTML preprocessing for the fragment of the Lycos query output shown in Figure 2. The example demonstrates a LOOP construct. The template corresponds to the first portion of the query results - see Figure 4 for the template that applies universally to all portions.

```
<set_var loop_start=1>
<loop>
<if loop_start = 0>
   . 
</if>
<set_var loop_start=0>
<if subcount = active>
  <FONT SIZE=+1><var subcount></FONT>
<else>
  <A HREF="/cgi-bin/pursuit?first=
    <var 10*(subcount - 1)+1>&part=&cat=lycos&
    query=<var query>"><var subcount></A>
</if>
</loop>
```

(a) The template.

```
query = caching+dynamic+objects;
active = 4;
<loop>
subcount = 1 to 10,1;
</loop>
```

(b) The bindings.

Figure 4: Example of conditional macro-expansion (the current page is 4).

last of the results are shown). The fragment can be obtained from the template and bindings shown in Figure 6.

2.2 Scoping

The scope of macro-variables is similar to scope in block-structured languages. Variables whose binding is specified within a loop in the DYNAMICS section have the scope limited to that loop. Variables that are bound outside any loop in the DYNAMICS section have global scope. They are shadowed by loop variables with the same name. Only a single binding can be specified to each global variable, and this binding is used everywhere this variable is encountered in the template. The order in which loop and global variables appear in the template loop is immaterial.

Loop variables have a list of values in their binding specifications. In an iteration, each variable is bound to the next value in its list, and the loop fragment of the template is expanded using these bindings. When the list is exhausted for some variable, the expansion stops, and the rest of all other lists is ignored.

The LOOP construct can be nested. This could be used to macro-encode a resource that contains a variable number of sections, each containing a variable number of similar entries. Consider, for example, a hypothetical service that gives restaurant information. A query could be: "Give me Chinese restaurants under $10 in Union County". The response lists results by a town. For each town, the resource provides general information (e.g., links to current events in the town or other categories of restaurants), and a list of Chinese restaurants. Thus, the resource contains a number of town entries, whose number varies with a query, with a variable number of restaurant entries within each town entry, again, depending on the query. One can easily provide examples of deeper loop nesting.

```
<FONT SIZE=-1 FACE=ARIAL, HELVETICA><b>1)
<a href="http://amsterdam.lcs.mit.edu/papers/www94.html"> Dynamic Documents </a>
</b></font><br><FONT size=-1 FACE=ARIAL, HELVETICA>
    Dynamic Documents: Extensibility and Adaptability in the WWW
    M. Frans Kaashoek, Tom Pinckney, and Joshua A. Tauber
    MIT Laboratory for Computer Science...</font>
<br><FONT SIZE=-1 FACE=ARIAL, HELVETICA>
    http://amsterdam.lcs.mit.edu/papers/www94.html
    [100%, 3 of 3 terms]</font><p>

<...>

<FONT SIZE=-1 FACE=ARIAL, HELVETICA><b>10)
<a href="http://fbp.icm.edu.pl/sunworldonline/swol-04-1996/swol-04-oobook.glossary.html">
    SunWorld Online distributed object glossary</a>
</b></font><br><FONT size=-1 FACE=ARIAL, HELVETICA>
     [Table of contents][Sun's homepage][Next story][Sidebar][Back to story]
    Glossary of Object-Oriented Terminology for Business Compiled by Mike Aube an...</font>
<br><FONT SIZE=-1 FACE=ARIAL, HELVETICA>
    http://fbp.icm.edu.pl/sunworldonline/swol-04-1996/swol-04-oobook.glossary.html
    [28%, 2 of 3 terms]</font><p>
```

Figure 5: Responses from the sample Lycos query.

When a template and the corresponding bindings contain multiple loops, some of which are nested, the correspondence between the loops in both parts is established structurally. For the top-level loops, the first loop in the template corresponds to the first loop in the bindings, etc. Within each top-level loop, the correspondence is established similarly for the next-level nested loops, and so on. It is an error to have non-matching loop structures of the template and bindings of the same resource.

One could provide further features of general macro-languages, like nested macros, where the binding to a variable contains a VAR-expression, or specifying full-blown macro-definitions, including macro-parameters, in the bindings. For example, when different resource representations are generated for different types of browsers (e.g., with or without support for frames) a single template can be cached at the proxy and conditionally expanded by different browsers without contacting the content provider.

3 HTTP and HTML Macro-expansion Implementation

While the previous section presented an overview of HTML macro-expansion, this section discusses issues involving HTTP and caching.

3.1 Methodology

The goal of HPP is to cache static content while permitting dynamic content to be transferred when needed. The browser would have to use an Accept-encoding request header to inform the server that an HPP encoding would be understood. In the steady state, a browser would have many commonly used templates in its cache, and receive a response header such as Content-encoding: x-hpp to describe dynamic data that has a reference to a template.

The question is, how should HPP handle a template that is *not* already cached? In the worst case, each x-hpp resource would refer to a template that is not already cached, and require a second round-trip to the content provider to obtain the template. Even with techniques such as the Keep-alive request header of HTTP 1.1 [5], which would use a single TCP connection to request both resources, the extra round-trip could dramatically increase the end-to-end latency to receive the original resource.

An alternative is to send the template along with the dynamic data as a MIME multipart document [7] when it is not cached already. This would require a mechanism for deciding when to send the template. One way to do this is to establish a one-to-one correspondence between a resource URL and an identifier (e.g., URL) of its template. This would let the client determine *a priori* if it has the template for a given URL in its cache, and then pass this information in an HTTP header, together with a version

```
<loop>
<FONT SIZE=-1 FACE=ARIAL HELVETICA><b><var counter>)
<a href=”<var queryurl>”> <var querysubj></a>
</b></font><br><FONT size=-1 FACE=ARIAL, HELVETICA>
    <var querysum></font>
<br><FONT SIZE=-1 FACE=ARIAL, HELVETICA>
    <var queryurl>
    [<var percent>%, <var nummatch> of <var numterms> terms]</font><p>
</loop>
```

(a) The template corresponding to the fragment of Lycos response shown in Figure 5.

```
<loop>
counter = 1 to 10, 1;
queryurl=
    ”http://amsterdam.lcs.mit.edu/papers/www94.html”,
    <...>
    ”http://fbp.icm.edu.pl/sunworldonline/swol-04-1996/swol-04-oobook.glossary.html”;
querysubj=
    ”Dynamic Documents”,
    <...>
    ”SunWorld Online distributed object glossary”;
querysum=
    ”Dynamic Documents: Extensibility and Adaptability in the WWW
    M. Frans Kaashoek, Tom Pinckney, and Joshua A. Tauber
    MIT Laboratory for Computer Science...”,
    <...>
    ”[Table of contents][Sun's homepage][Next story] [Sidebar][Back to story]
    Glossary of Object-Oriented Terminology for Business Compiled by Mike Aube an...”;
percent= 100, <...>, 28;
nummatch= 3, <...>, 2;
</loop>
```

(b) The bindings.

Figure 6: Example of a more complicated LOOP construct, with multiple explicit values per variable.

identifier of the cached template. (A version identifier, like a last-modified timestamp or an *etag* [5] is needed to ensure that the cached template is still current.) The downside of this approach is that it would reduce the flexibility of HPP. For instance, multiple resources might share a single template in the absence of this restriction. More importantly, some content providers generate significantly different output for the same URL, e.g., depending on the type of requesting browser. Thus, a URL may correspond to multiple templates.

The simplest way to address the above problems, and the one we take initially, is to include explicit template URL in bindings. The server first returns the bindings with the proper template URL, which is then fetched by the client unless cached. This approach assumes that templates will commonly be cached and that the added cost of a second request when a template is not already cached will be amortized over the benefits that otherwise accrue. A recently published study of Web accesses indirectly supports this assumption, showing an 85-87%

rate of repeated access to dynamic resources (see [12], Sections 5.1 and 5.2). If HPP reaches a wide enough distribution to gather meaningful statistics of real usage, it will be possible to determine how valid this assumption is in practice.

3.2 Comparison with other Techniques

HPP serves two purposes: to allow the dynamic portions of similar resources to be transferred without sending the static portions, and to allow a compact representation of repetition within a resource (through the use of the LOOP construct). In fact, both of these goals can be achieved through other means: the former via delta-encoding [1, 8, 12, 14] and the latter via compression [12, 13]. A fundamental issue with delta-encoding is the management of past versions, since a server and client must agree on a base version against which to apply a delta. The server may generate thousands of versions of a single dynamic resource and cannot easily store every past version that an arbitrary client might have stored. HPP addresses this problem by permitting a single cached template that all clients may cache, and providing additional dynamic data in the context of that template. A delta-encoding system could similarly notify clients that they should store a particular instance of a resource, against which future deltas would be computed [8]. Finally, delta-encoding does not help with repetition unless the encoding itself is compressed [12].

Simple compression of responses would eliminate redundancy from loops in a manner similar to HPP. In fact, compression could potentially achieve a better reduction in data size because it would compress all text, not just the obvious repetition from loops. However, as shown below, HPP templates and dynamic data can be compressed as well, getting the benefits of both compression and caching of the static portions.

Part of the rationale behind HPP is that it provides servers with the opportunity to generate *just* the dynamic data, rather than generating an entire HTML resource only to compute a delta-encoding or compressed version of it. The expected data transfer size in either case is comparable, but the server overhead should be less. Section 4 discusses performance issues.

3.3 Compatibility

HPP requires extra functionality from the client (to perform macro-preprocessing of templates) and the server (to recognize the x-hpp header from clients willing to accept the HPP encoding). Extra functionality can be added to clients without modifying existing browsers by co-locating a proxy with the browser [2]. CGI scripts on servers will have to understand how to generate HPP dynamic data, but that process can be automated via libraries or other tools.

HPP could also be implemented with no modification of client software using Java applets or as a plug-in. In the Java approach, when a client requests a dynamic resource, a page is returned containing the URL of a Java applet that implements HPP expansion and, as the parameters to the applet, the URL of the template and the bindings of the resource. The client would then fetch the applet (presumably, it would be cached in most cases since it is the same for all resources) and pass it the bindings and the template. The applet would then fetch the template and perform the expansion.

4 Performance Evaluation

To quantify the potential benefits of HPP, we have used it to encode dynamic resources in several contexts. Most of our experiments have been performed using a modified version of an internal Web-based "recruiting database." We chose this application for two reasons. First, it typifies the style of response that is well-suited to HPP: a query returns just a list of names that are hyperlinks to CGI invocations with the name of the candidate specified, or else returns a full display of all information about one or more candidates in a canonical form. Second, unlike search engines and other services both on the external Internet and elsewhere within the AT&T Intranet, we have full access to the CGI scripts, which we have modified to use HPP. In addition, to show broader applicability of HPP, we have encoded by hand some other dynamic resources, as described in Section 4.2.

4.1 Metrics

Useful metrics for evaluating HPP include user request latency, network bandwidth demands, and the load placed on content providers. We anticipate that templates will be generated as part of application development, using authoring tools similar to the ones used today for creating static documents. We assume that templates will be cached aggressively by clients; therefore, our evaluation focuses on the dynamic aspects of a query. On the server, we consider the overhead of generating the dynamic portion of a resource, compared to generating the entire resource as in traditional systems and then optionally computing a delta-encoded or compressed form. On the network, the total number of bytes transferred is an issue, though for small resources the round-trip time will dominate any bandwidth issues (i.e., reducing a 500-byte resource to a 200-byte resource will have minimal effect, but reducing a 5-Kbyte resource to a 2-Kbyte resource will be more noticeable). On the client side, reconstructing the original resource from HPP, a delta-encoded response, or a compressed response will add overhead in comparison to simply displaying an unencoded resource.

Mogul et al. [12] measured the performance of delta-encoding and compression on both the client and server, and found that a library-based encoding and decoding system (one that could be linked into an HTTP client or server, rather than invoked as a separate process) is significantly faster than a T1 line (193 KBytes/sec). For HPP, the combination of overhead and reduced bandwidth needs to be competitive with these other encodings for it to be practical. Our thesis is that by permitting an application to generate the dynamic data without dealing with the portions of the HTML resource that do not change between invocations, HPP encoding can be more efficient than generating the resource and then delta-encoding or compressing it. Also, the templates and bindings can themselves be compressed efficiently, for further size reductions.

Our experiments were conducted using Apache Web server running on a 200MHz Sun Ultra 1 workstation with 128 Mbytes of memory. For comparison with compression and delta-encoding, we used *vdelta*, the fastest known system for compression and delta-encoding [9].

4.2 Sample Data

In our preliminary experimental evaluation of HPP, we consider five HTML resources:

A The output of a query to the modified recruiting database (mentioned at the start of this section), listing 10 candidates by name only.

B The same query, listing candidates with full information.

C The same query as **B** but listing only one candidate.

D, E Queries to AltaVista [3] and Lycos [11] using the query string "caching dynamic objects."

4.3 Bandwidth Demands

Figure 7 shows the size of each resource using several formats: the unencoded resource, compressed using *vdelta*, delta-encoded using *vdelta*, and using HPP with and without *vdelta* compression. For the delta-encoding, we considered several possible base versions: a substantially similar resource, such as the first page of the response to a search engine query compared with the second page; a somewhat different response from the same search engine (searching for a different set of keywords); and the HPP template, which has much of the text that appears in the final resource. Here we present delta-encodings against the template, under the assumption that the template is similar to what a system like WebExpress [8] might do in designating a base version. Finally, in the case of HPP, the bars are broken into two components, with the lower bar indicating the dynamic resource and the higher one the template, which would presumably frequently be cached.

Purely from the standpoint of network bandwidth, when templates are cached, the size of the compressed HPP dynamic data is comparable to an efficient delta-encoding. The uncompressed dynamic data, though larger, is still significantly smaller than even the compressed original resource.

4.4 End-To-End Latency

Figure 8 shows end-to-end request latency using each type of encoding. For resources **A** through **C**, the figure reports actual latencies measured over 28.8K modem and averaged over 100 requests. Since we did not have a good implementation of the Web server that would incorporate compression and delta-encoding, the experiments with these encodings were conducted using pre-computed compressed and delta-encoded resources, so that the server would output the generated resource to /dev/null and read the appropriately pre-computed resource from a file to return to the client. Given that the overhead for encoding and decoding resources with *vdelta* is negligible (this overhead, shown later in Table 2, is always below 0.5% of the latency), it is omitted. We do include the HPP expansion overhead on the client, which ranges from 3 to 14% of the latency in our experiments.

For resources **D** and **E**, Figure 8 includes estimated transfer costs, based on the size of each resource, as well as measured HPP expansion costs from Table 2, using the following rationale. The transfer costs are estimated by using a fixed cost of 700ms for a connection set-up and a variable cost assuming the resource is transferred at 22Kbps. These parameters closely approximate measured latency for resources **A** and **B**. The measured latency for resource **C** was unexpectedly high; we are still investigating the reason for that. Again, these graphs do not include encoding and decoding costs for *vdelta*, because they would not be discernible. We also omit the overhead for generating the original resource and HPP dynamics because, first, measurements with the recruiting database scripts show that generating just the dynamic data is consistently slightly faster than generating the entire HTML resource, and second, AltaVista and

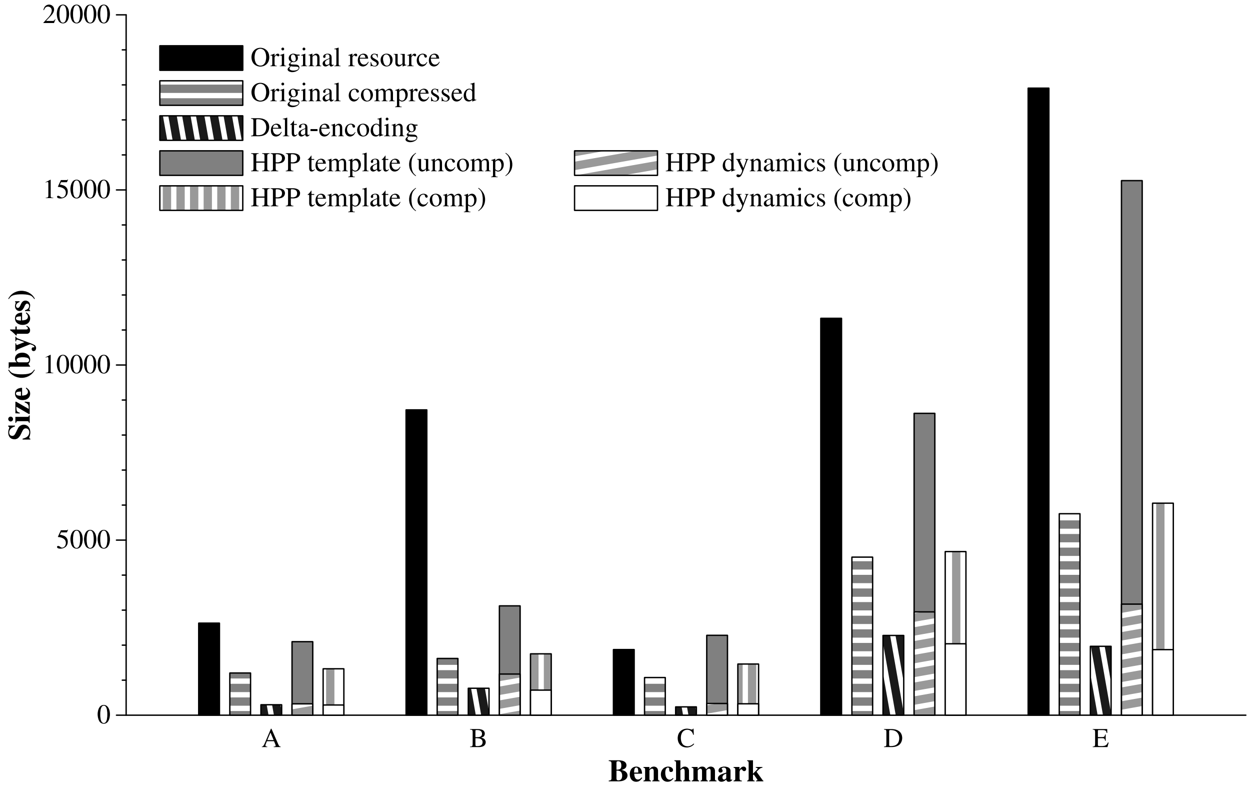

Figure 7: Sizes of the original and encoded resources for each query studied.

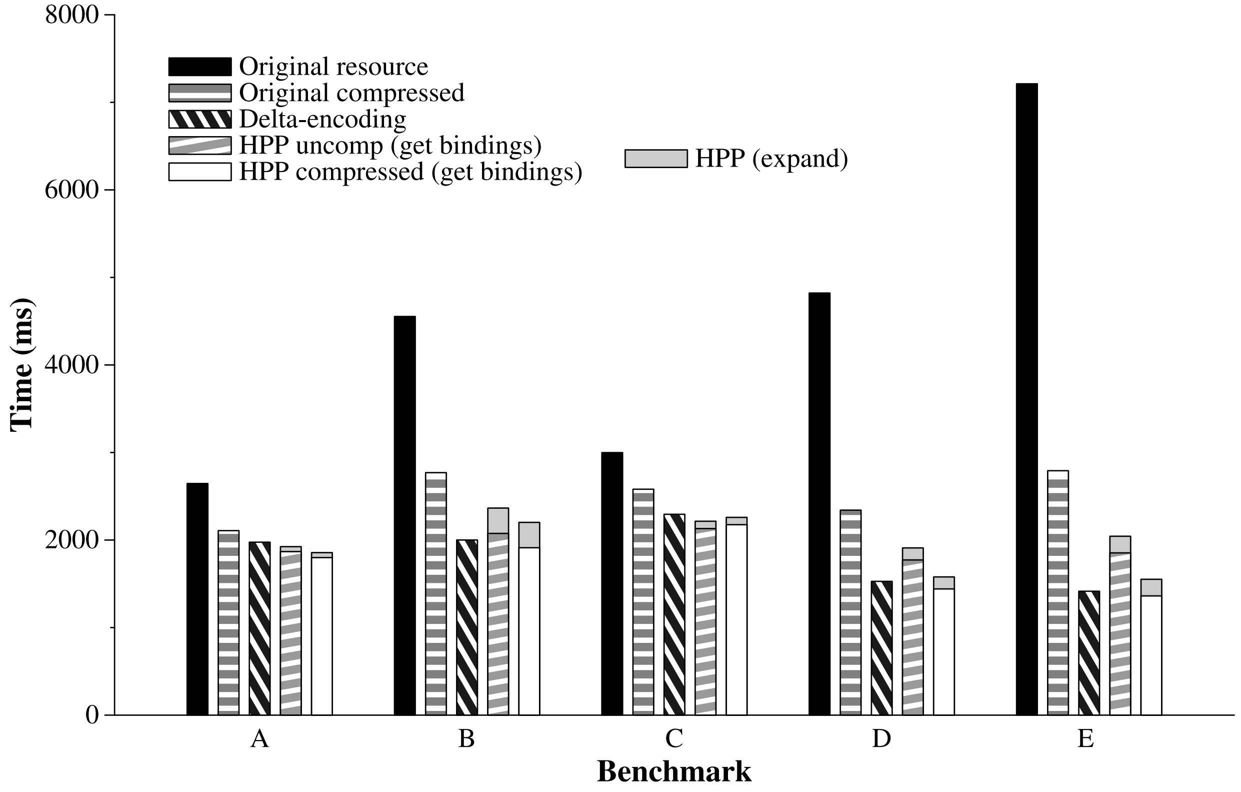

Figure 8: Measured or estimated end-to-end latency of the original and encoded resources for each query studied.

Resource	Full HML	HPP Dynamics
A	923.6ms	841.2ms
B	938.6ms	901.5ms
C	884.9ms	878.5ms

Table 1: Resource generation times (in ms) at the server.

Resource	Compress	Deltas	HPP expand
A	3.8	3.7	24.0
B	4.6	4.6	87.3
C	3.6	3.7	19.2
D	6.6	6.5	137.4
E	8.6	8.2	189.9

Table 2: Overheads for compression, delta-encoding, and HPP expansion (in ms).

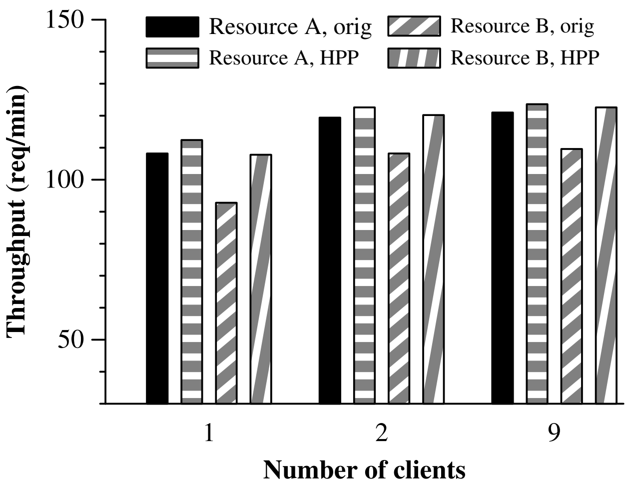

Figure 9: Measured server throughput.

Lycos are generated by extremely powerful machines so the latency should be dominated by a transmission over slow link. Also, without access to the Lycos and AltaVista CGI applications, we can only generate the dynamic data for these resources by hand.

Our measurements indicate that over a slow link, both delta-encoding and HPP can substantially reduce the end-to-end latency and overall network load. Compared to fetching uncompressed original resource, sending uncompressed HPP reduces the latency by between 26% for resource **C** and 72% for resource **D**, including the unoptimized macro-expansion on the client. For compressed resources, the latency reduction ranges from 12% to 44%, again including the expansion.

Comparing HPP and delta-encoding, they show essentially the same overall latency when taking into account HPP expansion on the client. If this expansion could be optimized to the level of *vdelta* overhead, HPP would result in between 4 and 9% less latency.

4.5 Server Load

Table 1 compares the time to generate the entire resource and just HPP dynamics. We could not include AltaVista and Lycos for resource and HPP generation without access to these applications. The results are averaged over 1000 invocations.

Table 2 gives the overhead for compressing original resources, computing *vdelta* with the template as the base version, and the HPP expansion time on the client. We timed *vdelta* and HPP encoding and decoding as the average of 100 loops within a single process that performed the operation; this method amortizes process startup costs and produces overhead comparable to a library implementation of an encoding and decoding system [12]. In addition, we have implemented the HPP expansion code as a "coprocess," again in order to amortize startup costs but to run in the context of Perl, with similar performance.

Table 1 shows that generating just the dynamic data is slightly but consistently faster than generating the entire resource. Computing and applying *vdelta* is always negligible compared to resource generation. The overhead for decoding compressed or delta-encoded resources on the client (not shown) were even slightly lower than the encoding costs. However, HPP decoding on the client is discernible and is an order of magnitude higher than *vdelta* overhead. One should note that HPP expansion is implemented as unoptimized Perl script. In a real implementation, HPP expansion overhead should be comparable to applying a delta-encoded resource against a previous version. Even if it is more expensive computationally than applying a delta, HPP would have the benefit of shifting load from servers to clients.

The lower time to generate HPP dynamics, compared to the entire resource, translates into increased server capacity, as shown in Figure 9. This figure compares throughput of the server using full HTML and HPP. In this experiment, client machines run multiple processes, with each process repeatedly sending a request to the server, waiting for the response, and immediately sending the next request. Client machines are connected to the server via 10Mbps Ethernet. The reported datapoints include one client machine running one process, two machines running one process each, and three machines with three processes per machine. The throughput is measured as the number of processed requests per minute in a five-minute experiment.

The results show that HPP consistently increases server throughput, by up to 15%. While scripts for the recruiting database are implemented with an unoptimized Perl library so it is difficult to generalize these results, they indicate that HPP may provide a sizable improvement in server capacity.

5 Related Work

There are several existing approaches that could be used to improve performance of access to dynamic pages. In the *optimistic deltas* approach of [1], a proxy cache optimistically transfers to the client a document that may be out-of-date, while obtaining the current version from the content provider. Upon obtaining the current version, the proxy sends to the client a (possibly empty) delta from the old version to the new version. If the client already has some stale version, it can include its version ID (e.g, an etag [5]) with its request, and if the proxy also has this version, it can send the delta only.

The optimistic deltas target mostly static pages when they expire in caches or are modified by the authors. Applying this mechanism to dynamic pages that are different for every access would require the content provider (or the proxy) to keep a large version pool, which would be used to compute the delta against the requesting client's version. We view optimistic deltas as complimentary to our approach: deltas could be used to improve access to templates when they are modified or expire.

The WebExpress system [8] allows the content provider to specify a base version of a dynamic document and then send delta-encodings against this version. It thus avoids the need for a version pool, since the end server always computes the delta-encoding against the base version. The advantage of WebExpress over HPP is that is does not require any changes to HTML. HPP, however, offers equivalent or better performance by avoiding the computation of the full HTML resource before computing a delta-encoding.

The W3C consortium has proposed an extension for HTML that would allow embedding arbitrary resources into the HTML resource [10]. A similar extension has been proposed by Netscape as well. A new OBJECT tag is introduced, which can be used to specify the URL of resources to be inserted as well as the URL of the code to interpret these resources. The code specification may be implicit based on the resource type. This mechanism could in principle be used to insert dynamic elements of the resource into a static template, as in HPP. However, each dynamic element must be treated separately - each must be requested, and the executable must be invoked each time to insert the next element into the resource. In addition, this method does not supports loops. The whole loop fragment has to be treated as a dynamic element. These loop elements constitute a large portion of typical resources, especially those returned by search engines. Thus, much of the benefits of HPP would not be realized.

6 Conclusion

We have proposed using macro-preprocessing at the client to support caching of dynamic HTML documents. In our approach, called HPP, HTML is extended to allow separation of the static and dynamic portions of a resource. The static portions (called the *template*) can then be cached, and only dynamic portions must be obtained on every access.

We described our HTML extension informally using real resources as examples. We also showed on the example of two popular search engines that our macro-encoding results in a remarkable reduction of the amount of data that must be fetched from the server on every access. The dynamic data is comparable in size to an efficient delta-encoding, especially if the dynamic data is itself compressed. Compared to the original resource, HPP bindings reduce the size by factors of 4–8 without compression, or 2–4 when comparing compressed bindings to the compressed original resource. The load of the server is also decreased since it does not have to transmit the template for clients that already have it in their caches. In fact, when data compression is used, the server load should decrease even for requests that miss in the client caches: templates, which constitute a large portion of resources, can be compressed once and served repeatedly at low cost. Given that data compression dramatically reduces network traffic and delays, this can become an important factor in the future.

HPP requires minimal changes to Web servers and no changes to the HTTP protocol. The changes on the server side are concentrated in CGI scripts and other applications that generate dynamic data. The changes on the client side can be provided in various ways, including a custom proxy co-located with the browser, a plug-in, a Java applet, or direct support within the browser.

Recent trace data [12] indicate that dynamic resources exhibit a rate of repeated access that is more than twice the rate of repeated access to static resources. Therefore, they promise a much higher hit ratio if cached. Allowing caching of these resources should result in disproportional gains in overall Web access performance.

Acknowledgments

Gideon Glass and Balachander Krishnamurthy provided helpful comments on earlier drafts. We also thank the

anonymous referees for their comments.

References

[1] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *Proceedings of 1997 USENIX Technical Conference*, pages 289–303, Anaheim, CA, January 1997. Also available as http://www.research.att.com/˜douglis/papers/optdel.ps.gz.

[2] Charles Brooks, Murray S. Mazer, Scott Meeks, and Jim Miller. Application-specific proxy servers as HTTP stream transducers. In *Proceedings of the Fourth International WWW Conference*, December 1995. Also available as http://www.w3.org/pub/Conferen=-ces/WWW4/Papers/56/.

[3] Digital Equipment Corporation. http://www.altavista.digital.com, January 1997.

[4] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: a live study of the World Wide Web. In *Proceedings of the Symposium on Internetworking Systems and Technologies*. USENIX, December 1997. To appear.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, et al. RFC 2068: Hypertext transfer protocol — HTTP/1.1, January 1997.

[6] International Organization for Standardization. Standard generalized markup language, 1986.

[7] N. Freed and N. Borenstein. RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part one: Format of Internet message bodies, December 1996.

[8] Barron C. Housel and David B. Lindquist. WebExpress: A system for optimizing Web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 108–116, Rye, New York, November 1996. ACM. Also available as http://www.networking.ibm.com/artour/artwewp.htm.

[9] James. J. Hunt, Kiem-Phong Vo, and Walter. F. Tichy. An empirical study of delta algorithms. In *IEEE Software Configuration and Maintenance Workshop*, 1996.

[10] Inserting objects into HTML. http://www.w3.org/pub/WWW/TR/WD-object.html.

[11] Lycos. http://www.lycos.com/, January 1997.

[12] Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of SIGCOMM'97*, pages 181–194, Cannes, France, September 1997. ACM. An extended version appears as Digital Equipment Corporation Western Research Lab TR 97/4, July, 1997, available as http://www.research.digital.com/wrl/techreports/abstracts/97.4.html.

[13] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of SIGCOMM'97*, pages 155–166, Cannes, France, September 1997. ACM.

[14] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for World-Wide Web documents. In *Proceedings of SIGCOMM'96*, volume 26,4, pages 293–305, New York, August 1996. ACM. Also available as http://ei.cs.vt.edu/˜succeed/96WAASF1/.