Server-side Caching Optimisations for the Apache Web Server

Simon Geard

October 24, 2000

Contents

List of Figures

List of Tables

Chapter 1

Server-side scripting

Most people are familiar with the idea of client-side scripting, where as well as static text, an HTML document also contains commands that are downloaded with the document, and processed by the browser. Javascript is the language normally used for this purpose, although others exist. Client-side scripting is usually used to add some degree of interactivity or animation to pages that would otherwise be static.

Server-side scripts are similar, except that they are run on the server. When a scripted document is requested, the server processes the script, and delivers the resulting output to the client. Because processing is done before the document is delivered, the scripts are invisible to the client - the delivered document is simply an HTML file, possibly containing client-side scripts, if these were generated by the server.

The most common use of server-side scripting is for database access. Security is greater, since the database does not have to be accessible across a wider network. Further, because the returned document does not contain the scripting commands, no information is revealed that would allow undesirable access to the database.

Almost every e-commerce system requires server-side scripting in some form, to generate catalogues from a database, or to process information sent by the user.

1.1 Scripting tools

There is a large variety of scripting tools available, depending in part on the choice of web server. Below is a discussion of some of the more common.

1.1.1 CGI programs

This is the traditional method for creating server-side applications. A CGI program is an external program, which is run by the server in response to client requests. Such programs can be written in almost any language; Perl, Python, C/C++, and shell-script are the most common.

The major limitation of CGI is that it lacks integration with the server. All of the other alternatives are implemented as modules within the server, giving them greater access to information, and greater control over the server.

1.1.2 PHP Hypertext Preprocessor

The PHP Hypertext Preprocessor (PHP) is a server-side scripting system developed as a module for the Apache web server for a Unix environment, although it can also be built as an executable to run under other servers and environments.

The PHP language is close to C++ or Java in syntax, although considerably simplified. Like C++, it is an object-oriented language, but objects are not integral as they are in Java.

A good example of what can be done with PHP is the Webmail system (*http://webmail.ec.auckland.ac.nz*) here at Auckland University - this uses an open-source application called IMP. Another is the vBulletin message board software, used by Infogrames (*http://www.ina-community.com*) for tech-support forums.

1.1.3 Active Server Pages

Active Server Pages (ASP) is the server-side scripting language provided by Microsoft. As implemented by Microsoft, ASP exists only for Microsoft web servers - Internet Information Server (IIS) and Personal Web Server (PWS). However, other companies (such as Chillisoft) have implemented ASP for other platforms, such as Apache.

Like most Microsoft scripting tools, ASP is based on Microsoft's VisualBasic language, so anyone familiar with VisualBasic, VBA, or other Windows scripting tools should have no trouble learning the language.

There are many sites that appear to use ASP (evidenced by a .asp extension), but do not actually use scripting features. However, there are several that do so - ASP Forums (*http://www.aspforums.com*) produce a variety of ASP-based web applications, including authentication systems, discussion forums, and a link-categorisation system similar to Yahoo. Another example is the Cecil system (*http://www.cecil.edu*) used at Auckland University.

1.1.4 Java Server Pages

Java Server Pages (JSP) is the system developed by Sun Microsystems, using Java Servlet technology. There are modules providing JSP support for both Apache and IIS servers, and several standalone JSP servers exist.

JSP has one interesting difference over other systems - it's compiled. While the page source remains in text format, the script is compiled into a servlet the first time it is requested. Unless the source changes, all other requests are satisfied by using the servlet, offering better performance for more complex scripts.

JSP is a relatively new technology, and isn't in widespread use yet. Even so, there are a number of sites making use of it to various degrees. Merge Technologies (*http://www.merge.com*) uses JSP throughout its site, but doesn't seem to have used any of the advanced features.

1.1.5 Server Side Includes

Server-Side Includes (SSI) is a server-side scripting system provided by Apache in the form of the add-on module *mod_include*. It's not exclusive to Apache, but that is the platform it is most often used on.

SSI is a very different system to the others, in that it does not provide a full programming language. Instead, it provides a series of directives which the parser replaces with the requested information, such as the value of an environment variable, or the contents of another file.

One site using SSI is Compuware Numega (*http://www.numega.com*). This company has used SSI documents for their entire site, though it's impossible to tell from the client side what kind of features they use.

Note: though SSI is the correct name for this system, it is often referred to as SHTML (Server-parsed HTML), probably because this is the standard file extension used by SSI documents. In this document, SSI refers to the technology, SHTML to documents using SSI.

1.1.6 Comparisons

SSI is the simplest scripting system, intended for a very different purpose to the other options. While the others are typically used for generating dynamic content from a data source, SSI is aimed towards creating static documents from static source documents - a typical use would be to keep navigation components independent from content, simplifying maintenance.

PHP, ASP and JSP are all very similar in terms of functionality, being complete programming languages with complex flow control and user-defined functions. All three have strong database support, and being server modules, offer some ability to control the server environment. ASP and JSP offer further functionality by allowing scripts to use COM and Java objects.

CGI can provide most of the same features, since such programs can do anything a regular application can do. However, because they run separately from the server, they do not have access to the server API and information tables. They also require the creation of a separate process, creating extra overhead.

1.2 SSI

As stated above, SSI is a very simple system. Instead of providing a complete programming environment, it provides a small set of directives, which the parser replaces with the requested content, or uses to alter the output of subsequent directives. Instead of running database queries for dynamic content, SSI simply parses these directives to combine static elements into static content.

Table 1.1 contains the complete list of directives supported by the SSI implementation used in the Apache server.

1.2.1 Examples

Throughout this report, a small set of documents is used as an example when one is needed. Though small, this example site represents a typical use of SSI for aiding in site maintenance. This section simply explains how each file is used - Appendix A contains the full SHTML/HTML source.

First are two top level documents, *foo.shtml* and *bar.shtml* - the files requested by the user. Both files contain a set of directives like those below:

config changes the appearance of certain types of output - the format of file size, and the format used for printing date/time values.

echo displays the value of an environment variable.

exec executes an external command.

fsize displays the size of a specified file.

flastmod displays the last-modified time-stamp of a specified file.

include inserts the contents of another file into this document. The included file is handled by Apache, and thus may be of any type that produces text output - plain text, HTML, or even another SHTML file.

printenv displays a list of all defined variables and their values.

set assigns a value to a variable.

if/elif/else/endif provide conditional display of sections of an SHTML document.

Table 1.1: List of SSI directives

```
<!--#set var="LOCATION" value="foo" -->
<!--#include virtual="header.shtml" -->
<!--#include virtual="foo-body.html" -->
<!--#include virtual="footer.shtml" -->
```

The first line creates an environment variable called LOCATION, and gives it the value "foo" (bar.shtml uses "bar" instead). This value is visible from this point on, including in child documents.

The remaining three lines identify child documents - each of these files is read by the server, and the output is embedded in the parent, replacing the *include* directive. If the child documents are SHTML files, they are also parsed, and so on, until all references are fully expanded.

The first of the child documents is *header.shtml*. This file contains directives used for constructing a simple menu of links, allowing the user to choose between Foo and Bar. The following code is used:

```
<!--#if expr="\"$LOCATION\" = \"foo\"" -->
  [ Foo ]
```

```
  [ <a href="bar.shtml">Bar</a> ]
<!--#elif expr="\"$LOCATION\" = \"bar\"" -->
  [ <a href="foo.shtml">Foo</a> ]
  [ Bar ]
<!--#endif -->
```

Basically, the value of LOCATION is used to determine which document the current request is for - recall that LOCATION is defined in the top-level documents. If the variable is set to 'foo', then the link to *foo.shtml* is disabled, indicating that this is the current document. Similarly, if LOCATION is set to 'bar', then the link to *bar.shtml* is disabled instead.

The aim is to simplify maintenance, by keeping the menu in a single file, instead of in each top-level document. This way, if new items are added, the only file that needs changing is *header.shtml* - updated menus automatically appear in the requested documents.

The last SHTML file, *footer.shtml* uses a simple trick for putting a standard footer at the bottom of every document, yet also including specific information about that document.

```
Last modified: <!--#echo var="LAST\_MODIFIED" -->
```

Very simple, the *echo* directive simply outputs the value of a variable. In this case, LAST_MODIFIED is a variable defined by Apache, and contains the date/time the requested file was last modified. Note that this is not the time-stamp for *footer.shtml*, but for the top level document.

Finally, there are two normal HTML files, *foo_body.html* and *bar_body.html*. Since these are not scripted files, they are handled directly by Apache, not by the SSI module. The result is the same though - the contents are embedded into the requesting document.

For this reason, HTML should be used in preference to SHTML where possible, since unscripted documents do not require parsing. For the standard SSI module, this will result in a reduced load on the server; for the caching version, it also reduces the amount of data to be stored, as unscripted files will not be cached separately.

Chapter 2

Caching SSI scripts

Under the existing implementation of SSI, documents are re-parsed every time a client requests them. While SSI scripts are fairly simple to parse, needing only a single pass, it is possible that re-parsing each document every time is not making the most efficient use of a web server.

Because SSI scripts are mostly static, it is possible to establish rules for determining whether the content of a document has changed. In this way, it is possible to cache parsed documents, and to construct a dependency-checking system for ensuring the validity of the cache.

2.1 Rules

The following list identifies the way that directives cause dependencies, and how each dependency can be resolved.

exec - Executes an external program. Because the output of such a program cannot be determined without running it, there is no way a script can be cached if it uses this directive.

printenv - Lists the current environment. Because the environment includes values such as the system time, it is not possible to cache this information.

fsize, flastmod - Displays the size and last-modified time-stamp of a specified file. This creates a dependency on that file, which must be tracked.

include - Includes the contents of another document within the current document. This creates a dependency on the file, and may also introduce further dependencies, if the child is another SSI script.

In addition, any directive may contain a reference to a variable. For example, *header.shtml* contains several references to the variable LOCATION, while *footer.shtml* refers to the variable LAST_MODIFIED. These references must also be tracked in order to ensure cache accuracy.

2.2 The caching process

Figure 2.1 shows the process involved in delivering a document. Essentially, if the document is in the cache, and if the cached copy is valid, the cached copy should be delivered. Otherwise, the cache must be re-generated.

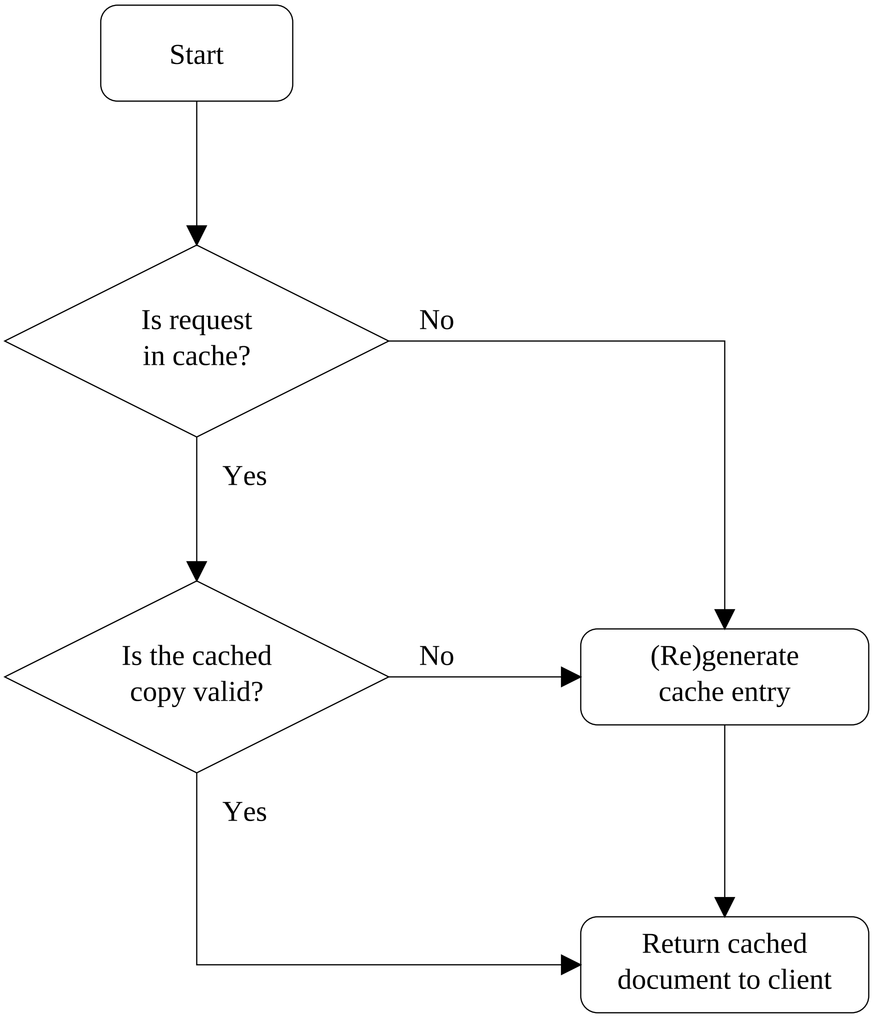

Figure 2.1: The caching process

2.3 Storage

The Apache environment is a dynamic one, where processes are created and destroyed frequently. Because of this, it is difficult to make efficient use of shared memory.

Instead, a purely disk-based system is used, where each SSI document corresponds to a file in a special directory. This ensures data persistence, and simplifies the sharing of data between processes. Disk caching comes at a cost, but may still offer performance improvements over re-parsing each component of a document.

For each document, there will be two files created. The first is the dependency list, tracking all of the dependencies for the current request. An example file would be *foo.shtml$dep*.

The second file is the cached data that is to be delivered to the client - the output from running this document through the parser. This file would be given a name such as *foo.shtml$cache*.

2.4 Tracking dependencies

There are three kinds of dependency which must be tracked, in order to ensure cache validity. The basic kind is a simple dependency on another file. The others are dependencies resulting from *includes* and dependencies resulting from references to variables.

Figure 2.2 shows the dependency tree for a request on the file *foo.shtml*. This tree would result in three dependency files being created, with the following names:

foo.shtml$dep

foo.shtml:header.shtml$dep

foo.shtml:footer.shtml$dep

2.4.1 File references

This is the most basic kind of dependency, being the leaves of the dependency tree. As long as infinite recursion does not occur (this is prevented by the module), all dependencies expand to file references.

foo.shtml$cache :	foo.shtml
	foo.shtml:header.shtml$cache
	foo-body.html
	foo.shtml:footer.shtml$cache
foo.shtml:header.shtml$cache :	foo.shtml
	header.shtml
foo.shtml:footer.shtml$cache :	foo.shtml
	footer.shtml

Figure 2.2: Example dependency tree for foo.shtml

File dependencies are simple references to a file - all that is necessary to check validity is to compare the time-stamp of the file with the time-stamp of the dependency file. As long as the latter is the more recently changed, the reference can be assumed to be fresh.

2.4.2 Included documents

If a sub-request is for another SHTML script, this document creates dependencies of it's own, and is treated separately. As Figure 2.2 shows, both *header.shtml* and *footer.shtml* have their own dependency entries.

If the target of an *include* is a normal HTML document, a simple file reference is all that is necessary. Since the content is perfectly static, a time-stamp check is all that is needed. Figure 2.2 shows how such files are treated in the dependency tree - only as a leaf of the parent, not as a separate branch.

However, if the target is another SHTML document, then another level of dependencies is created. As figure 2.2 shows, a branch is added to the dependency tree. This is what adds a recursive element to the dependencies.

Further, each child document is qualified with its parent. In the above examples, there is no entry for *header.shtml$dep*, since it will never be requested directly. Instead, there are separate entries qualified with the parent, called *foo.shtml:header.shtml$dep* and *bar.shtml:header.shtml$dep*.

2.4.3 Variables

Two types of variable are considered here - those created by *set* directives within scripts, and those defined by the Apache environment. These are resolved in different ways.

Variables defined within scripts can be handled by tracking which file in the hierarchy most recently assigned a value to that variable. With this information, the dependency can be reduced to a reference to that file. Since the variable cannot change without some change to the target file, a simple time-stamp check is all that is needed.

Environment-level variables are a different matter, since they are defined outside of the scope of the SSI module. Some of them are impossible to cache, since the value is determined only when the module runs. For example, UNIQUE_ID, or DATE_LOCAL. Table 2.1 contains a list of those defined on an Apache 1.3.12 system, as provided by the *printenv* directive.

The solution will be to handle them on a case-by-case basis. A small number of them, for example, identify the requested file. These can be handled by adding dependencies as appropriate. In Table 2.1, these would include variables such as SCRIPT_FILENAME, DOCUMENT_URI, DOCUMENT_NAME, and LAST_MODIFIED.

Since the remaining variables cannot be tracked using file dependencies, any reference to these variables will mean that the document will not be cached.

DOCUMENT_ROOT	/home/httpd/htdocs
HTTP_ACCEPT	*/*
HTTP_ACCEPT_CHARSET	iso-8859-1,*,utf-8
HTTP_ACCEPT_ENCODING	gzip
HTTP_ACCEPT_LANGUAGE	en
HTTP_CONNECTION	Keep-Alive
HTTP_HOST	localhost
HTTP_USER_AGENT	Mozilla/4.7 [en] (X11; I; Linux 2.2.5-15 i686)
PATH	/sbin:/usr/sbin:/bin:/usr/bin:/usr/X11R6/bin
REMOTE_ADDR	127.0.0.1
REMOTE_PORT	1161
SCRIPT_FILENAME	/home/httpd/htdocs/test.shtml
SERVER_ADDR	127.0.0.1
SERVER_ADMIN	delgarde@geocities.com
SERVER_NAME	infinity.home
SERVER_PORT	80
SERVER_SOFTWARE	Apache/1.3.12 (Unix) PHP/3.0.16
UNIQUE_ID	OTtM3cCoAAwAAAJHBHQ
GATEWAY_INTERFACE	CGI/1.1
SERVER_PROTOCOL	HTTP/1.0
REQUEST_METHOD	GET
QUERY_STRING	
REQUEST_URI	/test.shtml
SCRIPT_NAME	/test.shtml
DATE_LOCAL	Monday, 05-Jun-2000 18:46:53 NZST
DATE_GMT	Monday, 05-Jun-2000 06:46:53 GMT
LAST_MODIFIED	Thursday, 01-Jun-2000 22:07:11 NZST
DOCUMENT_URI	/test.shtml
DOCUMENT_PATH_INFO	
USER_NAME	root
DOCUMENT_NAME	test.shtml

Table 2.1: List of Apache-defined variables

Chapter 3

Implementation

Where chapter 2 developed an abstract design for caching SSI scripts, this chapter focuses on issues that had to be resolved in order to implement that design - in particular, the platform to develop for, and the means of storing the cache data.

3.1 Platform

The caching module development is aimed at the Unix platform. Although Apache supports others – such as Windows, OS/2, and VMS – Unix is the environment Apache is most frequently used under. Further, the availability of Linux and associated development tools made it the ideal environment for developing and testing Apache code.

The next platform decision is the version of Apache, since two development families are currently in progress. The first is the 1.3 family, which has been around for several years, and is in widespread use. The second is the newer 2.0 family, which is a major rewrite and redesign, with extensive modifications to the API and module system.

The original decision was to use the 1.3 family. Being older, the code for this family is much more stable - the code has been thoroughly tested, and is unlikely to undergo significant changes. Since 2.0 is still only a developers release, it seemed that to use this family would be to invite problems.

However, during development, it was discovered that 1.3 lacked a critical feature needed for this project - the ability to redirect the output of a module. While it would be possible to work around this limitation, it would require extensive changes to the entire module, and would still not be ideal.

As a result, the decision was made to switch to the 2.0 family instead - the current release at the time being 2.0alpha6. The previous development on 1.3 was quickly ported to 2.0, with only minimal difficulty.

Unfortunately, the feature needed - called Layered IO by the Apache developers - is still very experimental, and prone to change. As implemented in 2.0a6, it wasn't suitable for the purpose. Discussion with the developers indicated that the 2.0alpha7 implementation would be ideal, but alpha7 was released too late to be of use, and once released was found to be somewhat bug-ridden.

The final decision was to revert to Apache 1.3, and accept the less elegant and effective solution of matching each function call writing to the screen with one writing to disk.

It remains possible to create a 2.0a7 based implementation, once released. Combining work done on 2.0a6 with some experimentation of 2.0a7, all of the necessary code exists, so once a debugged version is available, applying the changes should be trivial. Unfortunately, it will not be possible to include this version as part of the project.

3.2 Filenames

The previous chapter included a simplified version of the cache file naming system. However, that version has no way of representing paths, treating all files as though they are in an assumed directory. For the actual implementation, a more complex scheme is needed for file naming. Three possible solutions are discussed here.

3.2.1 Recreate directory structure

With this solution, a set of subdirectories would be created, matching the location of the source document. Thus, if the requested document is located at */usr/local/apache/htdocs/path/to/foo.shtml*, then the cached copy of the document would be at */usr/local/apache/cache/path/to/foo.shtml*.

This approach has the advantage of being easy to read, since only the base path is changed. However, it also adds complexity, by requiring extra code to maintain the directory structure - something that is undesirable during development.

3.2.2 Encode file names

Under this solution, a file identifier is encoded in some way, allowing an entire path to be stored as a single filename. Thus, a request for a document with the URI */path/to/foo.shtml* might have special characters encoded in such a way to produce the identifier *%2fpath%2fto%2ffoo%2eshtml*.

This approach is very simple to implement, and the resulting identifiers are still human-readable, with only minimal effort. The main disadvantage is that once identifiers are combined to reflect file *includes*, there is the danger that the filename may be longer than the file system will allow. The maximum length varies between file systems, but 256 characters is typical.

3.2.3 Hashing file names

Hashing is a variant of the encoding solution, but instead of encoding individual characters, a hash function is used to convert an entire string into a unique numeric value. Thus, the request */path/to/foo.shtml* might produce the identifier *de9c40b3* if run through a 32-bit hash function.

This approach eliminates the problem of file names being too long, and is fairly easy to implement, given a suitable hashing algorithm. However, the hashing process is not reversible, and is not human readable - a fact that makes testing difficult.

3.2.4 Decision

During development, filename encoding will be used, using the method described above. It is essentially the same scheme used by the HTTP protocol for encoding special characters - replacing the character by a percentage sign, followed by the hexadecimal value of the ASCII character.

This solution was chosen because it's simple to implement, and because the resulting identifiers are human-readable. During development, testing will be done using a small set of pages, so the file system limitations are not likely to pose a problem. If the module were to be released for general use, it may be necessary to switch to one of the alternatives - probably hashing.

Figure 2.2 showed the simplified version of the file structure. Figure 3.1 shows the same table, modified to use the encoding scheme. It assumes that all files are stored in the server root directory, such as */foo.shtml*.

In the figure, two types of dependency are clearly visible - encoded identifiers, and full path names. The presence of an identifier means that there

```
%2ffoo%2eshtml :
        /apache/htdocs/foo.shtml
        %2ffoo%2eshtml$%2fheader%2eshtml
        /apache/htdocs/foo-body.html
        %2ffoo%2eshtml$%2ffooter%2eshtml

%2ffoo%2eshtml$%2fheader%2eshtml :
        %2ffoo%2eshtml
        /apache/htdocs/header.shtml

%2ffoo%2eshtml$%2ffooter%2eshtml :
        %2ffoo%2eshtml
        /apache/htdocs/foo.shtml
```

Figure 3.1: Example dependency tree for /foo.shtml

is another entry that needs to be handled. This will be the case for file *includes*, where parent depends on child, while the child depends on other files.

If a dependency is not on an identifier, then it contains the full path to a file. This shows a simple dependency, where the given file has either changed, or not changed.

Since infinitely recursing *includes* are not allowed, all entries will eventually expand to a set of simple references. Figure 3.2 shows the fully resolved dependency list for */foo.shtml*. All identifiers have been expanded in the recursion process, leaving only simple file references.

```
%2ffoo%2eshtml :
    /apache/htdocs/foo.shtml
    /apache/htdocs/header.shtml
    /apache/htdocs/foo-body.html
    /apache/htdocs/footer.shtml
```

Figure 3.2: Resolved dependency list for /foo.shtml

Chapter 4

Implementation stages

Early in development, a plugin for Microsoft's IIS server was discovered that provided caching of ASP scripts. This plugin is a much simpler design than the design outlined in the Chapter 2, but is aimed at the same area - simple scripts that will change only rarely.

Comparing the two designs, it seemed that the ASP plugin could be considered a step along the way, since the changes needed to create such a plugin for SSI are a subset of the changes needed for the full design.

For this reason, development of the SSI caching module has been split into three stages, each of which is a usable module on its own, and be compared with the others during performance testing.

4.1 Stage One: Semi-automatic caching

The first stage was to provide functionality equivalent to that provided by the ASP plugin referred to above. The design is not identical, however, since the ASP plugin is implemented as a separate filter, not as a change to the ASP scripting system itself.

The logic used in the ASP filter, and in Stage One is simple. If a certain file exists in a special directory, that file should be delivered directly to the client. Otherwise, the source SHTML file should be parsed and delivered.

In this version, there is no concept of dependency checking - there is no way of determining whether the cached copy is valid or not. Since generating the cached file is also out of the scope of the module, it is possible that this content of this file is totally different from the output of the source.

The following code shows the changes made to the basic module. Utility functions aren't shown, only those that illustrate the logic used.

```
if (is_cache_valid(r, cfg)){
    send_cached_page(r, cfg);
}
else{
    send_parsed_content(f, r);
}
```

In the main handler, the following condition is inserted. *send_parsed_content()* is the original function for parsing scripts, and is unchanged in Stages 1 and 2. *send_cached_page()* is a new function which reads a cache file and delivers it to the client. In the standard module, the if/else block is not present - *send_parsed_content* is a;ways called.

The third function used, *is_cache_valid()* contains the logic which determines whether or not the cached copy should be used. In this version of the module, the function is simple:

```
static int is_cache_valid(request_rec *r, ci_config *cfg){
    /* Get dependency name */
    request_id = get_id(r);
    cache_file = apr_pstrcat(
        r->pool, cfg->cache_path, "/", request_id, "$cache", NULL);
    /* Check that cache file exists. */
    if (stat(cache_file, &finfo) == -1)
        return 0;
    else
        return 1;
}
```

4.2 Stage Two: Full dependency checking

Stage Two is an extension to Stage One, providing the full dependency checking mechanism described in Chapter 2. This means that for each document, there should now be a second file in the cache directory, containing dependency information. This file would contain a list of dependencies, similar to the entries shown in Figure 3.1, where sub-requests and normal references are easily distinguished by the first character - a '%' or a '/'.

In code, most of the functionality has moved from *is_cache_valid()* to a new function, *check_id()*. This is a recursive function for checking dependency files. Essentially, it reads a dependency file line by line, and for each line, takes appropriate action.

If the line represents a file, the time-stamp of that file is checked, returning 0 (false) if the file is more recent than the cache. If the line represents a sub-request, the function calls itself to process the new dependency file.

A table is used to record whether an entry has been checked, in order to prevent infinite recursion if several dependency files form a loop.

```
static int check_id( const char *req_id, request_rec *r,
                     table *table, svr_config *cfg ){
    /* Record this request_id as being checked */
    ap_table_set( table, req_id, "DONE" );
    /* Get filenames */
    cache_file =
        ap_pstrcat( r->pool, cfg->cache_path, req_id, "$cache", NULL );
    dep_file =
        ap_pstrcat( r->pool, cfg->cache_path, req_id, "$dep", NULL );
    /* First check - does a dependency file exist? */
    if(stat( dep_file, &finfo1 ) == -1 ) return 0;
    /* Second check - does a cache file exist? */
    if(stat( cache_file, &finfo1 ) == -1 ) return 0;
    /* Check contents of dependency file */
    f = ap_pfopen( r->pool, dep_file, "r" );
    fd = fileno( f );
    len = read( fd, buffer, BUFSIZ );
    while( index < len ){
        /* If end of line, check line */
        if( buffer[index] == '\n' ){
            buffer[index] = 0;
            /* If line is a subrequest entry and
                hasn't been handled already, recurse */
            if( *str == '%' ){
                if( ap_table_get( table, str ) == NULL )
                    if( check_id( str, r, table, cfg ) == 0 )
                        return 0;
            }
            /* Else check file timestamp here */
            else{
                if( stat( str, &finfo2 ) == -1 ) return 0;
                if( finfo2.st_mtime > finfo1.st_mtime ) return 0;
            }
            str = buffer + index + 1;
        }
```

```
        else index++;
    }
    ap_pfclose( r->pool, f );
    /* No failures, so return true */
    return 1;
}
```

The *is_cache_valid()* function is reduced to doing some initialisation work, then making the initial call to *check_id()*. No other changes have been made, so in this stage, both the dependency and cache files still must be created and maintained manually.

4.3 Stage Three: Fully automated caching

Stage Three is the completed module described in Chapter 2. Stage Two provided the complete dependency checking mechanism, but was not capable of generating cache or dependency files itself. Stage Three adds that missing ability.

Validity checking

The validity checking mechanism is identical to that used in Stage 2 - using the recursive *check_id* function to look at all appropriate dependency files to determine whether or not the cache can be used.

Dependency generation

For the generation of dependency files, two items were needed. Firstly, the list of dependencies to be written to file. A simple linked list was used, along with a handful of functions for adding items to the list.

At each point in the code where a dependency is created, a call to *add_dep_entry()* is made, adding the new dependency to the list. Once processing is finished, another function iterates through the list, writing to the dependency file for the request.

The second thing needed was to track where variables were being used, as discussed in chapter 2. In the standard module, an environment table is maintained for each request, tracking the value of each variable. To track these variables, a second table was added, recording where a variable was most recently set.

Variables are only created at two points in the code - *handle_set()* which is called when a *set* directive is encountered, and *add_ include_vars()* which creates the initial environment. At both points, code was added to maintain a duplicate variable in the second table, containing the id of the appropriate file.

Similarly, variables are only accessed in two points. The *if* and *elif* directives make use of the function *parse_expr()* which expands variables as part of a regex parser. All other directives use the *parse_string()* function, which performs much the same role.

In both cases, the following code is used. If the location of the variable is recorded, that location is added to the dependency table. Otherwise, caching this file is declared impossible:

```
file = ap_table_get(r→notes, var);
if (file) {
    add_dep_entry(dep_table, file);
}
else {
    set_no_cache(dep_table);
}
```

Cache generation

Though conceptually simple, this is the feature that has caused the most problems. In order to be effective, it's necessary to use the existing code for output, but to somehow redirect the output to a file instead of to the client.

Failing this, it would be necessary to change every line of code that dealt with output, forcing it to write to a file instead. Worse, it would also be impossible to *include* any file that was handled by another module - a limitation which would make the module pointless, since most SSI-based sites make use of this ability.

Unfortunately, while the 2.0alpha7 release provides the needed ability, it wasn't available in time to be useful. For this reason, cache generation has been implemented as a 'hack' based on the 1.3 code.

The implementation is simple, if inefficient. In the top-level function *send_parsed_file()*, a file is opened for writing, and a file pointer passed to all functions that perform output. At each call to an Apache output function, a second call is added, writing the same output to file.

```
char *tmp = ap_escape_html(r->pool, val);
ap_rputs(tmp, r);
fputs(tmp, cf);
```

The function *ap_rputs()* is one of several Apache API functions for sending output to the client. The *fputs()* line is the one added, taking the same string and writing it to the file pointer *cf*.

This is complicated by sub-requests. Since there's no way to trap the output of the sub-request, it's necessary to take the cache file produced by the sub-request, and copy the contents into the parents cache file.

Chapter 5

Testing

Two issues of testing were involved. During the implementation stage, testing that the module worked correctly was the main concern. After that, performance testing became important, requiring some reworking of code to improve results.

5.1 Validity testing

Validity testing was essentially a manual process - making changes to the source pages, and checking by eye that the output was correct.

For Stage 1, very little complexity is involved. Either a cached copy of the document exists, or it doesn't. Testing it was simply a matter of checking that the cache document was being used if present.

For Stage 2, complete dependency checking is performed, providing more functionality to test. In order to fully test the code outlined in chapter 3, the following checks needed to be made:

- Cache must not be used if:
  - A dependency file does not exist.
  - A cache file does not exist,
  - Referenced files are more recent than cache,
- Sub-requests must follow the above rules.
- Module must not recurse infinitely if dependencies form a loop.

Testing these involved manually creating and deleting cache and dependency files, and editing the source documents to cause a change in timestamp.

Stage 3 involved the same tests as Stage 2, but in this case, the cache and dependency files are maintained automatically. As a result, two extra checking needed to be made - testing the accuracy of the generated dependency files, and checking that cache files were being updated and used.

5.2 Performance testing

Initially, a custom-made program called *wcat* was used to compare the performance of the different module versions. The program acted like a multi-threaded client, hitting the server with simultaneous requests for a set of SSI pages. The code for *wcat* is in Appendix B.

It turned out, however, that Apache already provides a program *ab* for benchmarking. While *wcat* was perfectly adequate, *ab* is more configurable, and produces a clearer output. As a result, *ab* was used to gather the measurements used in this report. The key statistic produced is the number of requests being processed per second - this being the standard indicator of web server performance.

5.2.1 Tests

At this stage, four different modules exist – the basic non-caching module, and the three caching versions. For the caching modules, the cache and dependency files were created in advance (using stage 3). For all tests, these files were 'fresh' - they would pass the tests used by the Stage 2 and 3 dependency checking.

For each of the caching modules, the test was performed 3 times, assuming a 0%, 50%, and 100% cache hit rate. Rather than alter the source files to cause cache invalidation, a random element was introduced to the code. With this element in place, the actual validity of the cache is tested, but the result is ignored. This simplifies testing considerably, while remaining a fair test.

Because of the nature of the dependency checking mechanism, there is some variability in the tests for Stages 2 and 3. This is due to the fact that a dependency can be failed with a single function call, but must pass all tests to be accepted. This is discussed in more detail further on.

5.2.2 Results

For each module and cache hit rate, the test was repeated 5 times. The data shown below represents an average of the 5 runs.

	Cache hit rate		
Module	0%	50%	100%
Basic	300		
Stage One: Semi-automatic caching	281	401	625
Stage Two: Full dependency checking	251	364	544
Stage Three: Fully automated caching	162	281	543

As Figure 5.1 shows, the Stage 1 module produces the best results. Because of the minimal overhead involved in checking for a cache file, very little cost in incurred when the cache can't be used - even for a 0% hit rate, the performance is within 10% of the basic module. For a 100% hit rate, performance is more than doubled.

Stage 2 shows the greater overhead of the dependency checking functions. In the chart, this is fairly constant, with performance about 10% lower at all hit rates. In practise, the overhead should be lower at lower hit rates, because of the nature of the testing - one failed test allows the cache to be rejected, while all tests must be passed for the cache to be used. This doesn't show up here, due to the manner testing was performed - a full test was performed, but the results ignored.

Additional testing of this, for a 0% hit rate, produced results ranging from 250 requests per second for a full dependency check, up to 280 for the most simple check. In the latter case, the test involves nothing more than a test for the existence of the dependency file - essentially the same as the Stage 1 test. Results reflect this - performance in this case is identical to Stage 1.

Stage 3 has the overhead of Stage 2, plus the cost of generating the cache and dependency files. At low hit rates, this difference is substantial - at 0%, Stage 3 completes 35-40% fewer requests than Stage 2. This difference narrows as hit rates increase. At 100%, Stage 3 never has to re-generate the files, and correspondingly, performance is identical to that of Stage 2.

Stage 3 has the same testing issue as Stage 2. However, the time-cost of performing a full test is considerably smaller than that of generating files, so the difference is less pronounced.

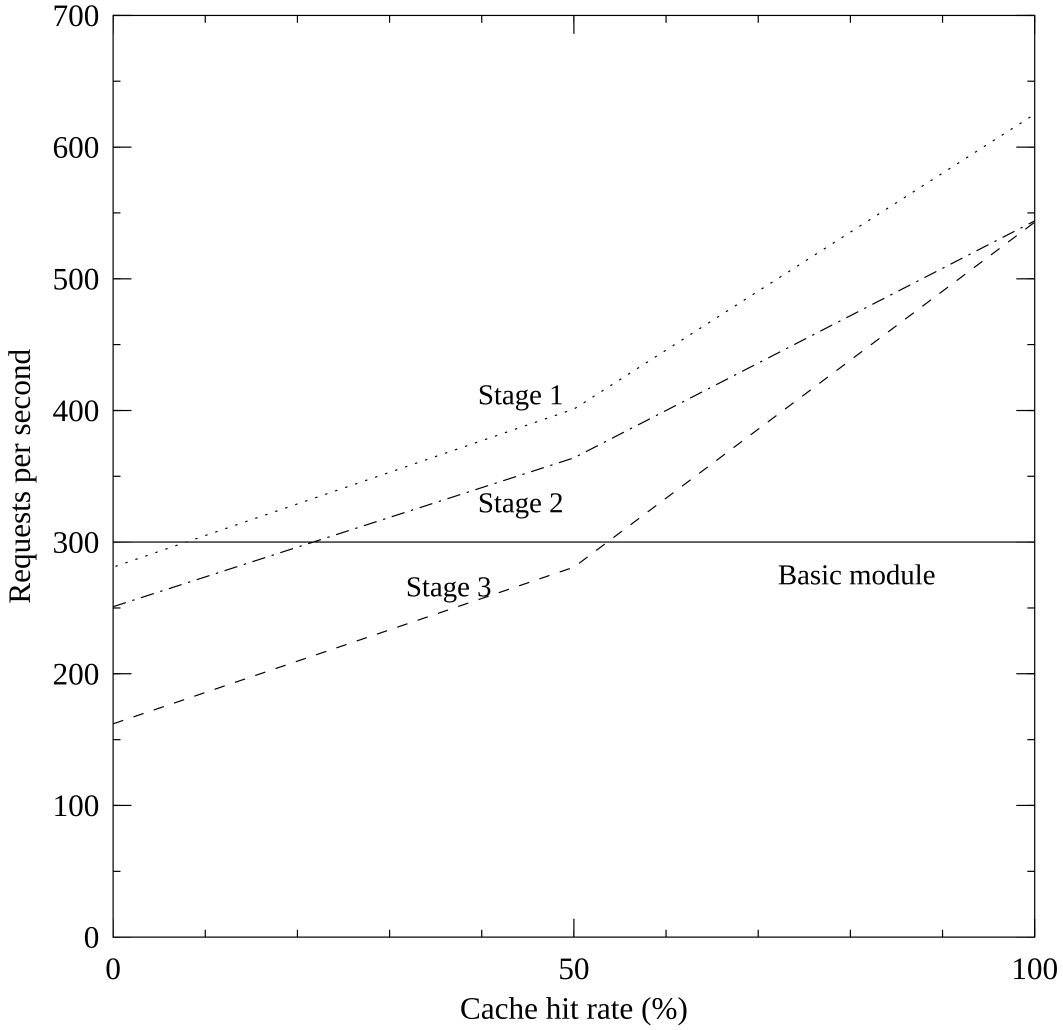

Figure 5.1: Performance comparisons

Chapter 6

Client-side caching

While the aim of this project is to investigate *server-side* caching of scripts, it also opens up the possibility for caching on the client side.

6.1 Implementation

As implemented in HTTP, client-side caching works via modification dates. If a document is suitable for client-side caching, the server returns an extra header, "Last-Modified". This header contains the date/time the document was last changed.

If a client has received such a header and cached the document, it can then pass that date back to the server in an "If-Last-Modified" header, next time it requests the document. If the document has not changed, the server simply returns a "304 Not Modified" response code, and does not bother sending the content.

Normally, it would not be possible for scripts to use this facility, since there is no way to determine when the content of a complex document last changed. However, the server side mechanism adds this ability, by ensuring that the cached data must be at least as recent as its most recent component.

Given that, supporting client-side caching is simple. Apache provides functions for setting a "Last-Modified" header, and for checking this header against the clients "If-Modified-Since" header. The following code, taken from the main handler, demonstrates the ability.

```
if( (req_time = is_cache_valid( r, cfg )) != -1){
    /* Can use client side cache? */
    ap_update_mtime( r, req_time );
    ap_set_last_modified( r );
```

```
    if( (errstatus = ap_meets_conditions(r)) != OK )
        return errstatus;
    /* No, must use use server side */
    /* Send headers */
    ap_send_http_header(r);
    /* Send content */
    send_cached_page( r, cfg );
}
else{
    ...
}
```

The first change is a modification to the dependency checking functions. Instead of returning 0/1 for a fail/pass, they now return the time-stamp of the top-level cache, or -1 for a failed test. This obtains the needed time-stamp without the cost of extra function calls.

The first two lines add a "Last-Modified" header to the response. The third calls a function that compares that header with the clients "If-Modified-Since" header, and if appropriate returns a "Not-Modified" response.

Otherwise, headers must be sent, followed by the content. Normally, the call to *ap_send_http_headers()* would be earlier in the handler, but it was moved to fit in with the caching mechanism.

Testing that these extensions worked required making manual requests, via a telnet connection to the server. Browsers are fine for checking output validity, but are designed to hide caching from the user. Manually checking responses is the only way to check if the mechanism is working.

6.2 Performance

The necessary changes were only made to Stage 1 and 3 modules; Stage 2 was skipped since it has no real value on it's own. Testing assumed a 100% cache hit rate on the server. On the client side, 4 situations were used - "If-Modified-Since" headers that could be satisfied 0%, 50%, and 100% of the time, plus the situation where no header was sent.

Testing was under the same conditions as before, using the Apache Benchmark tool. As before, each test was repeated 5 times, and an average taken.

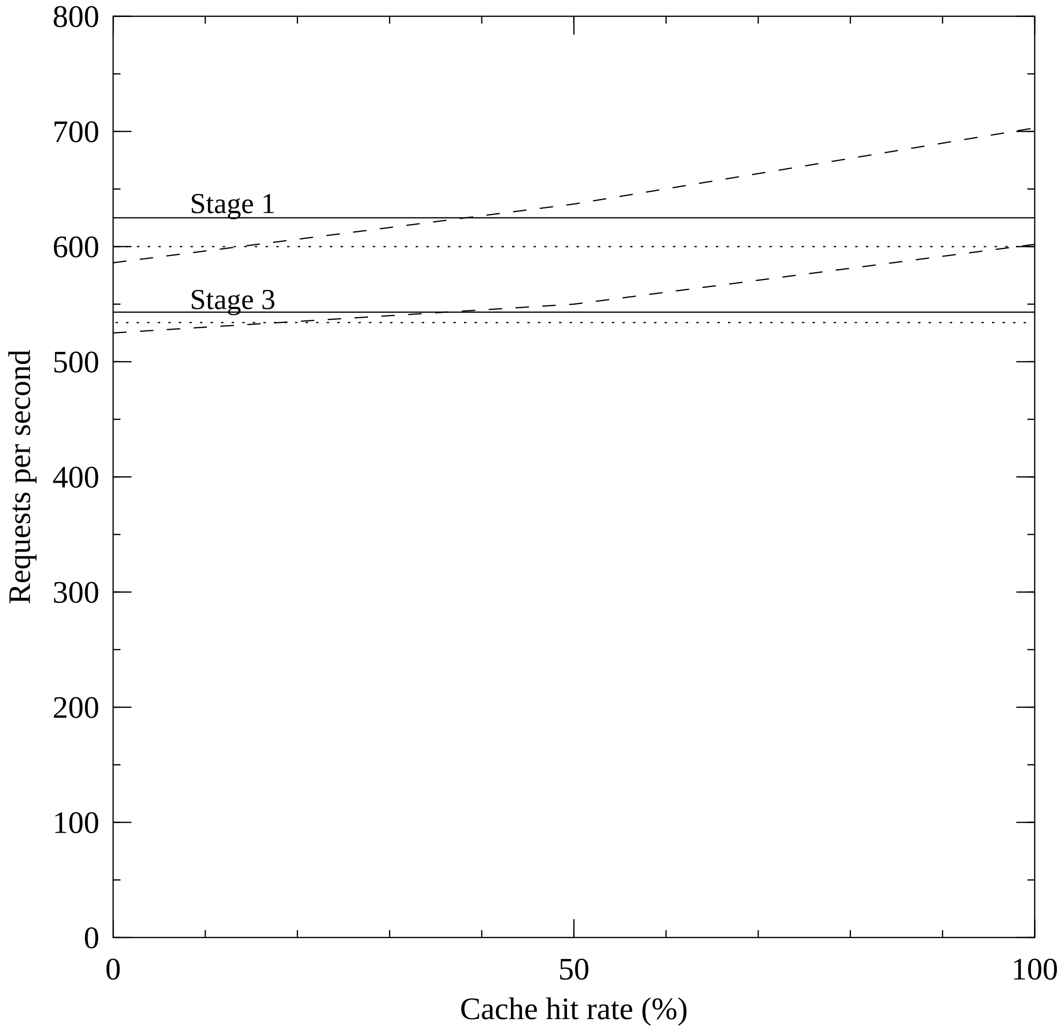

Figure 6.1: Performance comparisons for client-side caching

Module	Cache Hit Rate 0%	50%	100%	None
Stage One: Semi-automatic caching	586	638	703	600
Stage Three: Fully automated caching	525	550	602	534

Figure 6.1 displays two distinct sets of data, for Stage 1 and Stage 3. The solid horizontal lines represent the performance level produced by the unchanged module, without support for client side caching.

The dotted lines show the situation where caching support is added but not used; i.e where no "If-Modified-Since" header is sent. This is the situation for all new visitors to a page - data will not be in the cache, so the client will not send the header. Performance is a little bit below that of the base version, because of the extra overhead of checking the headers.

Finally, the dashed lines show the situation where a client tries to use the cache. Where the cache can't be used, there's slightly more overhead than for the previous case, since a full comparison is a little more complicated. This is the same story as for server-side dependency checking - the cache may be rejected quickly if it simply doesn't exist, but it takes more effort if time-stamps must also be checked.

When the client's copy can be used, there's a small gain over the basic module of about 10-15%. Not as great as that achieved by adding server-side caching, but still significant.

6.3 Conclusions

The most significant difference between server-side and client-side caching is the typical hit rates. For server-side, the cached data is in one place, and can be used to satisfy requests from any number of clients. As a result it's quite possible to achieve cache hit rates close to 100%.

This isn't the case for client-side caching. To achieve even a 50% hit rate, it's necessary that on average, every visitor to a page returns at least once before the page is next changed. For sites where a large proportion of visitors are unlikely to return frequently, client-side caching costs more than it saves.

This isn't the full story though. Processing time isn't the only scarce resource for a web server - network capacity is another. By minimising the amount of data being sent, client-side caching reduces the load on a network. This aspect couldn't be tested, but depending on specific cases, it may be that the savings are sufficient to make it worth the effort.

Chapter 7

Conclusions

From the results displayed in chapter 5, it's clear that the server-side caching extensions allow a considerable improvement in server performance. Above a 50% cache hit rate, all three stages operate above that of the basic, non-caching module - Stage 1 manages more than double the number of requests if the hit rate is near 100%, and the more complex Stage 3 achieves an 80% gain under the same conditions.

For typical uses, this is a reasonable situation. As discussed in chapter 1, the typical use of SSI aimed at simplifying web sites by separating common code for menus, headers and footers into separate files. SSI is not suited to dynamic content, which means files are unlikely to change often. If this is the case, achieving near-100% hit rates is simple - for a page changing once a day, 100 requests a day is enough to give a 99% hit rate.

Stage 2 has no real value beyond this project - it's an intermediate stage that matches the lesser functionality of Stage 1 with the lesser performance of Stage 3. For this reason, I won't be taking it any further now that the project is complete.

Stages 1 and 3, in contrast, offer quite a lot of potential. Stage 3 offers the complete functionality described in chapter 2. It offers a good performance boost, and runs completely without maintenance. With further optimisation (discussed in chapter 8), performance might be boosted to near that of Stage 1.

As is, Stage 1 offers considerably better performance than Stage 3, but because it can't tell when documents change, something else must take responsibility for making sure the cache is accurate. For now, the person responsible for maintaining pages would have to manually reconstruct the cache whenever documents change. However, it would not be difficult to automate the process, so that running a single program is the only thing necessary.

If this level of manual intervention is acceptable, the gains are great. By putting some of the functionality elsewhere, Stage 1 eliminates the need for dependency checking, and the need to generate cache and dependency files. This gives a greater performance, at only minimal cost.

Worth noting is the fact that this is the approach chosen for the ASP filter discussed earlier. Instead of allowing the module run automatically, caching every document it could, both Stage 1 and the ASP filter require the user to decide which documents are suitable for caching. The ASP filter also provided tools for manually storing content in the cache.

7.1 General caching

SSI caching was not the only aim of this project. Beyond that, the goal was to study the merits of server-side caching in general, using SSI as the basis for that study.

The following list identifies the key factors that determine whether or not server-side caching is appropriate to a given case.

- Time cost of generating content
- Time cost of checking cache
- Time cost of generating cache and dependency data
- Time cost of delivering cached content
- Expected cache hit rate

The expected hit rate is the most obvious. If the cached content can rarely be used, there is little point in suffering the costs associated with the caching mechanism. The rate at which caching becomes worthwhile is determined by the other factors - the time cost of various actions.

Firstly, the cost of generating the content. This is a property of the scripting system, and will vary considerably. SSI is simple to process, and thus has a relatively low time cost. In contrast, full 'languages' like ASP or PHP have a much greater complexity, and thus a greater time cost.

Second, the cost of determining whether or not cached content can be used. This shows up clearly at low hit rates, where performance is less than that achieved without the overhead of cache support. This cost is heavily dependent on the complexity of the dependency checking mechanism; more complex checking may be more accurate, but adds noticeably to overhead.

Third, the cost of generating the dependency and cache files. This is going to be affected by two things - the complexity of the dependency-checking mechanism, and the size and complexity of the document. Documents making heavy use of *includes* will be a lot more costly than single-level documents, even if the size is the same.

Fourth, the cost of delivering the cached content. If code is efficiently written, this cost should be fairly insignificant compared to the others, and is affected only by the quantity of data to be retrieved.

Finally, there's the expected cache hit rate. This is the factor that balances the various costs, to decide whether the costs of implementing the cache outweigh the cost of generating content for each request. As SSI caching showed, there is a point where the costs balance. As long as the hit rate is above this level, performance is enhanced by using the caching extensions. Below that point, performance is reduced, as the cost of providing caching is greater than the cost of not providing it.

Chapter 8

Future work

Despite the positive results achieved, there are several further improvements that can be made. In particular, there is still the possibility for better performance. Secondly, once stable 2.0 releases are available, it's desirable to apply the work to the newer server, since it is likely to replace 1.3 quickly once complete.

8.1 Optimisation

During early testing, all stages showed poor performance, even at high cache hit rates. To fix this, a lot of effort was put into optimising two critical parts of the code - the dependency checking functions *check_id()* and *is_cache_valid()*, and the *send_ cached_page* function for delivering data from the cache.

Adding memory buffering to both functions produced a considerable performance gain. Un-optimised, even the Stage 1 module never managed to exceed the performance of the basic module. That compares with the almost doubled performance after optimisation.

Despite this, there is still room for some improvement. While Stages 1 and 2 cannot be much improved, Stage 3 is considerably slower than Stage 2 because of the overhead of writing to the cache. Since this code was written only as a hack in the absence of Apache 2.0a7, no real effort was made to improve performance.

As implemented, Stage 3 writes to a cache file whenever it writes to the screen. The way this works out, a lot of small writes are being made, sometimes as little as a few bytes. If the other optimisations are anything to go by, buffering this output and writing it in larger chunks should considerably improve the performance at low hit rates.

Further, the basic module itself has some room for improvement. This is changed in 2.0a7, but in previous releases, source files are read character at a time. Again, adding memory buffering could improve performance at low hit rates.

8.2 Apache 2.0alpha7 implementation

Looking at the new design, re-implementing the code for the alpha7 release should be an improvement, particularly for Stage 3. Cache generation in the 1.3 code is fairly clumsy, and even with buffering added, it's not a clean solution.

The key feature in Apache 2.0 is Layered IO, which enables one module to receive the output of another. As implemented in alpha7, this consists of a chain of filters, connected with a pipe-like IO structure. At one end of the chain, the source file is read; at the other end, the output of the chain is sent to the user.

The basic SSI module is an example of a filter. It reads input from some previous filter in the chain, processes SSI directives, and sends the output to the next filter.

As planned, a Stage 3 implementation would work by adding a second filter to the chain, immediately after the SSI filter. While modifications to the existing filter would determine whether to use cached or fresh output, the new filter would be responsible for trapping the output and writing it to file. The code sample below shows what the second filter should look like.

```
static int cincludes_filter(ap_filter_t *f, ap_bucket_brigade *b){
    request_rec *r = f->r;
    const char *cache_file = "/home/delgarde/testfile";
    /* Value should be determined by looking
     * at a flag in r->notes, or something similar */
    int must_save;
    if( apr_table_get( r->notes, "##CACHE##" ) )
        must_save = 1;
    else
        must_save = 0;

    /* If cache needs creating, copy all buckets to a file */
    if( must_save ){
        ap_bucket *e;
        apr_file_t *cf;
```

```
        apr_status_t rv;
        /* Open output file */
        apr_remove_file( cache_file, r->pool );
        rv = apr_open( &cf, cache_file, APR_WRITE | APR_CREATE,
                       0x666, r->pool );
        if( rv != APR_SUCCESS ){
            return rv;
        }
        /* Iterate */
        e = AP_BRIGADE_FIRST(b);
        while( e != AP_BRIGADE_LAST(b) ){
            const char *str = NULL;
            apr_ssize_t n = 0;;
            /* Read */
            e->read(e, &str, &n, 0 );
            /* Write */
            apr_write( cf, str, &n );
            /* Move to next */
            e = AP_BUCKET_NEXT(e);
        }
        /* Close file */
        apr_close( cf );
    }

    /* Pass the brigade to the next filter */
    ap_pass_brigade( f->next, b );
    return APR_SUCCESS;
}
```

Simply, if the flag "##CACHE##" is set by the previous filter, this filter reads all the data, and echoes it to file. Otherwise, the output is passed straight to the next filter without being touched. According to Ryan Bloom - one of the developers responsible for the Layered IO mechanism - this should be the most efficient way of trapping the output.

Note: this code has not been tested, since a stable release of alpha7 was not available at the time this report was written. The design has been approved by Ryan, but may contain bugs that testing would reveal.

Appendix A

Example pages

This appendix contains SHTML and HTML documents used as examples throughout this report.

A.1 foo.shtml

```
<html>
 <body>
  <h1>Welcome to Foo</h1>
  <!--#set var="LOCATION" value="foo" -->
  <!--#include virtual="header.shtml" -->
  <!--#include virtual="foo-body.html" -->
  <!--#include virtual="footer.shtml" -->
 </body>
</html>
```

A.2 foo-body.html

```
<p>This is the content from foo_body.html</p>
```

A.3 bar.shtml

```
<html>
 <body>
  <h1>Welcome to Bar</h1>
  <!--#set var="LOCATION" value="bar" -->
  <!--#include virtual="header.shtml" -->
  <!--#include virtual="bar-body.html" -->
  <!--#include virtual="footer.shtml" -->
 </body>
</html>
```

A.4 bar-body.html

```
<p>This is the content from bar_body.html</p>
```

A.5 header.shtml

```
<div>
<!--#if expr="\"$LOCATION\" = \"foo\"" -->
[ Foo ]
[ <a href="bar.shtml">Bar</a> ]
<!--#elif expr="\"$LOCATION\" = \"bar\"" -->
[ <a href="foo.shtml">Foo</a> ]
[ Bar ]
<!--#endif -->
</div>
```

A.6 footer.shtml

```
<hr>
Last modified: <!--#echo var="LAST_MODIFIED" -->
```

Appendix B

WCAT source

```
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
#include <string.h>


const int NUM_OF_RUNS = 5;
const int NUM_OF_THREADS = 5;
const int REQS_PER_THREAD = 10000;

const char *HOST = "localhost";
const int PORT = 8080;
const int NUM_URIS = 15;
const char *URIS[ ] = {
  "/shtml/foo.shtml",
  "/shtml/bar.shtml",
};

/*
 * Host address, looked up in advance
 */
struct sockaddr_in addr;

/*
 * One running thread, connecting to a server.
 */
```

```
void *thr_run( void *param ){
  //Make repeated requests
  for( int r=0; r<REQS_PER_THREAD; r++ ){
    //Connect to server
    int sock = socket( AF_INET, SOCK_STREAM, 0 );
    if( connect( sock, (sockaddr *)&addr, sizeof(addr) ) == -1 ){
      fprintf( stderr, "Connection failed\n", r );
      return NULL;
    }
    //Make the request
    char buffer[1025];
    snprintf( buffer, 1024, "GET %s HTTP/1.0\n\n",
          URIS[r%NUM_URIS] );
    write( sock, buffer, sizeof(buffer) );
    //Get response
    buffer[1024]=0;
    while( int len = read( sock, buffer, 1024 ) );
    //Close connection
    close( sock );
  }
  return NULL;
}


/*
 * Application function
 */
int main( int argc, char **argv ){
  //Lookup host address
  struct hostent *host = gethostbyname( HOST );
  addr.sin_family = AF_INET;
  addr.sin_port = htons( PORT );
  memcpy( (char*)&addr.sin_addr.s_addr, host->h_addr_list[0],
host->h_length );
  //Make repeated runs
  for( int run=0; run<NUM_OF_RUNS; run++ ){
    printf( "Run %d:  Starting\n", run );
    //Get starting time
    int start_time = time(0);
    //Create info table
    pthread_t threads[NUM_OF_THREADS];
    //Create threads
    for( int t=0; t<NUM_OF_THREADS; t++ )
      pthread_create( &threads[t], NULL, &thr_run, NULL );
```

```
    //Wait for threads to terminate
    for( int t=0; t<NUM_OF_THREADS; t++ )
      pthread_join( threads[t], NULL );
    //Get end time
    int end_time = time(0);
    printf( "Run %d:  Total time %d seconds\n", run,
end_time-start_time );
  }
}
```

Appendix C

Stage 3 source

This appendix contains the code for the Stage 3 implementation, without the client-side caching extensions. Full code is available from the project web site, at http://www.tcs.auckland.ac.nz/~sgea003

All of the major additions to the module are in separate functions – all of which are grouped together at the top of the file, and marked with comments. Further changes are in the *send_parsed_file()* handler, and are also marked.

Because of the way this was implemented, a lot of small changes had to be made, and these are not marked. In most cases, these are no more than single function calls, or extra parameters passed to existing functions.

```
/* ====================================
 * Copyright (c) 1995-1999 The Apache Group. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in
 * the documentation and/or other materials provided with the
 * distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 * software must display the following acknowledgment:
```

* "This product includes software developed by the Apache Group
* for use in the Apache HTTP server project (http://www.apache.org/)."
*
* 4. The names "Apache Server" and "Apache Group" must not be used
* to endorse or promote products derived from this software without
* prior written permission. For written permission, please contact
* apache@apache.org.
*
* 5. Products derived from this software may not be called "Apache"
* nor may "Apache" appear in their names without prior written
* permission of the Apache Group.
*
* 6. Redistributions of any form whatsoever must retain the following
* acknowledgment:
* "This product includes software developed by the Apache Group
* for use in the Apache HTTP server project (http://www.apache.org/)."
*
*THIS SOFTWARE IS PROVIDED BY THE APACHE GROUP
*"AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES,
*INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
*WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
*PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT
*SHALL THE APACHE GROUP OR ITS CONTRIBUTORS BE
*LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
*SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
*(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
*SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
*OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
*CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER
*IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
*NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF
*THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
*POSSIBILITY OF SUCH DAMAGE.
* ====================================
*
* This software consists of voluntary contributions made by many
* individuals on behalf of the Apache Group and was originally based
* on public domain software written at the National Center for
* Supercomputing Applications, University of Illinois, Urbana-Champaign.
* For more information on the Apache Group and the Apache HTTP
* server project, please see <http://www.apache.org/>.
*
*/

```
/*
 * http_include.c: Handles the server-parsed HTML documents
 *
 * Original by Rob McCool; substantial fixups by David Robinson;
 * incorporated into the Apache module framework by rst.
 *
 */
/*
 * sub key may be anything a Perl*Handler can be:
 * subroutine name, package name (defaults to package::handler),
 * Class->method call or anoymous sub {}
 *
 * Child <!--#perl sub="sub {print $$}" --> accessed
 * <!--#perl sub="sub {print ++$Access::Cnt }" --> times. <br>
 *
 * <!--#perl arg="one" sub="mymod::includer" -->
 *
 * -Doug MacEachern
 */

#ifdef USE_PERL_SSI
#include "config.h"
#undef VOIDUSED
#ifdef USE_SFIO
#undef USE_SFIO
#define USE_STDIO
#endif
#include "modules/perl/mod_perl.h"
#else
#include "httpd.h"
#include "http_config.h"
#include "http_request.h"
#include "http_core.h"
#include "http_protocol.h"
#include "http_log.h"
#include "http_main.h"
#include "util_script.h"
#endif

#define STARTING_SEQUENCE "<!--#"
#define ENDING_SEQUENCE "-->"
#define DEFAULT_ERROR_MSG "[an error occurred while
processing this directive]"
#define DEFAULT_TIME_FORMAT "%A, %d-%b-%Y %H:%M:%S %Z"
```

```
#define SIZEFMT_BYTES 0
#define SIZEFMT_KMG 1
#ifdef CHARSET_EBCDIC
#define RAW_ASCII_CHAR(ch) os_toebcdic[(unsigned char)ch]
#else /*CHARSET_EBCDIC*/
#define RAW_ASCII_CHAR(ch) (ch)
#endif /*CHARSET_EBCDIC*/

module MODULE_VAR_EXPORT includes_module;
#define DEFAULT_CACHE_PATH "/home/delgarde/apache13/cache"

/* ========================================
 * Additional functions below
 * ======================================== */


/* ————————– Configuration function ————————–- */

/*
 * Configuration structure
 */
typedef struct svr_config{
   char *cache_path;
} svr_config;

/*
 * Return the appropriate configuration record.
 */
static void *get_server_config( server_rec *s ){
   return ap_get_module_config( s->module_config, &includes_module );
}
static void *get_directory_config( request_rec *r ){
   return ap_get_module_config( r->per_dir_config, &includes_module );
}
static void *get_request_config( request_rec *r ){
   return ap_get_module_config( r->request_config, &includes_module );
}

/*
 * Create server configuration entry.
 */
static void *create_includes_svr_config( pool *p, server_rec *s ){
   svr_config *cfg = ap_pcalloc( p, sizeof(svr_config) );
   cfg->cache_path = DEFAULT_CACHE_PATH;
```

```
    return cfg;
}

/*
 * Handle CacheIncludePath directive.
 * Sets the path containing caching information.
 */
static const char *set_cache_path(cmd_parms *cmd, void *conf, const
char *arg){
    char buffer[10];
    /* Get config */
    svr_config *cfg =
        get_server_config( cmd->server );
    /* Check path */
    arg = ap_os_canonical_filename( cmd->pool, arg );
    if( !ap_is_directory( arg ) )
        return "CacheIncludePath must be a valid directory";
    /* Copy value */
    cfg->cache_path = ap_pstrdup(cmd->pool, arg );
    /* Return if no error */
    return NULL;
}

/* ------------ Utility functions ------------ */

/*
 * Take the given path specifier, and encode all
 * non-alphanumeric characters in %xx format.
 */
static char *encode_path( pool *p, const char *path ){
    char output[MAX_STRING_LEN];
    const char *in_ptr = path;
    char *out_ptr = output;
    while( *in_ptr != 0 ){
        if( isalnum( *in_ptr ) )
            *out_ptr++ = *in_ptr;
        else
            out_ptr += sprintf( out_ptr, "%%%x", *in_ptr );

        in_ptr++;
    }
    *out_ptr = 0;

    return ap_pstrdup( p, output );
```

```
}

/*
 * Take a request, and generate a name that uniquely
 * identifies this request (including parent requests
 * if applicable.)
 */
static char *get_id( request_rec *r ){
   request_rec *parent = r;
   char *id = encode_path( r->pool, r->uri );
   /* Walk upwards */
   while( parent = parent->main ){
      char *parent_id = encode_path( r->pool, parent->uri );
      id = ap_pstrcat( r->pool, id, ":", parent_id, NULL );
   }
   return id;
}
/*
 * Get dep and cache filenames.
 */
static char *get_cache_file( request_rec *r, svr_config *cfg ){
   char *req_id = get_id( r );
   return ap_pstrcat( r->pool, cfg->cache_path, "/", req_id, "$cache",
NULL );
}
static char *get_dep_file( request_rec *r, svr_config *cfg ){
   char *req_id = get_id( r );
   return ap_pstrcat( r->pool, cfg->cache_path, "/", req_id, "$dep",
NULL );
}

/*
 * Get timestamp of given file.
 */
static time_t get_stamp( const char *filename ){
   struct stat finfo;
   if( stat( filename, &finfo ) == -1 ) return -1;
   return finfo.st_mtime;
}

/*
 * Check whether the given request_id can be taken
 * from the cache, or whether it needs regenerating.
 */
```

```
static int check_id( const char *req_id, request_rec *r,
                          table *table, svr_config *cfg ){
   struct stat finfo1, finfo2;
   char *dep_file, *cache_file;
   char buffer[BUFSIZ], *str = buffer;
   int index, fd, len;
   FILE *f;
   /* Record this request_id as being checked */
   ap_table_set( table, req_id, "DONE" );
   /* Get filenames */
   cache_file =
      ap_pstrcat( r->pool, cfg->cache_path, "/", req_id, "$cache", NULL );
   dep_file = ap_pstrcat( r->pool, cfg->cache_path, "/", req_id, "$dep",
NULL );
   /* First check - does a dependency file exist? */
   if(stat( dep_file, &finfo1 ) == -1 ) return -1;
   /* Second check - does a cache file exist? */
   if(stat( cache_file, &finfo1 ) == -1 ) return -1;
   /* Check contents of dependency file */
   f = ap_pfopen( r->pool, dep_file, "r" );
   fd = fileno( f );
   len = read( fd, buffer, BUFSIZ );
   while( index < len ){
      /* If end of line, check line */
      if( buffer[index] == '\n' ){
         buffer[index] = 0;
         /* If line is a subrequest entry and
              hasn't been handled already, recurse */
         if( *str == '%' ){
            if( ap_table_get( table, str ) == NULL )
               if( check_id( str, r, table, cfg ) == -1 )
                  return 0;
         }
         /* Else check file timestamp here */
         else{
            if( stat( str, &finfo2 ) == -1 ) return -1;
            if( finfo2.st_mtime > finfo1.st_mtime ) return -1;
         }
         str = buffer + index + 1;
      }
      else index++;
   }
   ap_pfclose( r->pool, f );
   /* No failures, so return true */
```

```
    return finfo1.st_mtime;
}


/*
 * Take a request, and decide whether the cache can
 * be used, or if document must be regenerated.
 *
 * For stage #2, full dependency checking is done,
 * recursively calling the check_id function.
 */
static int is_cache_valid( request_rec *r, svr_config *cfg ){
    char *request_id;
    table *table;
    /* Get dependency name */
    request_id = get_id( r );
    /* Create storage table */
    table = ap_make_table( r->pool, 50 );
    /* Check it recursively */
    return check_id( request_id, r, table, cfg );
}

/*
 * Send the cached copy of the requested document.
 * Note - no error detection - must be fixed.
 */

static void send_cached_page( request_rec *r, svr_config *cfg ){
    FILE *f;
    char *cache_file;
    char buffer[BUFSIZ];
    int len=0, fd=0;
    /* Get filename and open it */
    cache_file = get_cache_file( r, cfg );
    f = ap_pfopen( r->pool, cache_file, "r" );
    fd = fileno( f );
    /* Echo contents */
    while( (len = read( fd, buffer, BUFSIZ )) >0 ){
        ap_rwrite( buffer, len, r );
    }
    /* Close file */
    ap_pfclose( r->pool, f );
}
```

```
/*
 * Copy given file to a given file pointed.
 */
static void copy_cached_page( request_rec *r, FILE *cf, svr_config *cfg){
   FILE *f;
   char *cache_file;
   char buffer[BUFSIZ];
   int index, counter=0;
   char char_value;
   int len, fd;
   /* Get filename, and open it */
   cache_file = get_cache_file( r, cfg );
   f = ap_pfopen( r->pool, cache_file, "r" );
   fd = fileno( f );
   /* Dump contents */
   while( (len = read( fd, buffer, BUFSIZ )) >0 ){
      fwrite( buffer, 1, len, cf );
   }
   /* Close file */
   ap_pfclose( r->pool,f );

}


/* ----------- Dependency tracking ----------- */

/*
 * Simple structure for storing dependencies.
 */
typedef struct dep_list_t{
   const char *item;
   struct dep_list_t *next;
} dep_list_t;

typedef struct dep_table_t{
   int can_cache;
   pool *p;
   request_rec *r;
   dep_list_t *first;
} dep_table_t;

/*
 * Create a dependency table.
 */
```

```
static dep_table_t *make_dep_table( request_rec *r ){
   dep_table_t *dep_table = ap_palloc( r->pool, sizeof(dep_table_t) );
   dep_table->r = r;
   dep_table->p = r->pool;
   dep_table->can_cache = 1;
   dep_table->first = NULL;
   return dep_table;
}

/*
 * Record a new dependency
 */
static void add_dep_entry( dep_table_t *table, const char *name ){
   dep_list_t *i, *n;
   int t;
   if( !strcmp( name, get_id( table->r ) ) ) return;
   /* Create entry */
   n = ap_palloc( table->p, sizeof(dep_list_t) );
   n->item = ap_pstrdup( table->p, name );
   n->next = NULL;
   /* If list empty, assign first */
   if( table->first == NULL ){
      table->first = n;
      return;
   }
   /* Else find a place for it */
   i = table->first;
   while(1){
      if( !strcmp( i->item, name ) ) return;
      else if( i->next == NULL ) break;
      i = i->next;
   }
   i->next = n;
}

/*
 * Flag request as uncacheable.
 */
static void set_no_cache( dep_table_t *table ){
   ap_rputs( "Can't cache\n", table->r );
   table->can_cache = 0;
}

/*
```

```
 * Write dep table to file
 */
static void write_dep_table( dep_table_t *table, const char *file){
  dep_list_t *i;
  FILE *f = ap_pfopen( table->p, file, "w" );
  for( i=table->first; i!=NULL; i=i->next )
    fprintf( f, "%s\n", i->item );
  ap_pfclose( table->p, f );
}

/* ========================================
 * Additional functions above
 * ======================================== */

/* ----------- Environment function ----------- */

/* XXX: could use ap_table_overlap here */
static void add_include_vars(request_rec *r, char *timefmt)
{
#ifndef WIN32
    struct passwd *pw;
#endif /* ndef WIN32 */
    table *e = r->subprocess_env;
    table *n = r->notes;
    char *t;
    time_t date = r->request_time;

    ap_table_setn(e, "DATE_LOCAL", ap_ht_time(r->pool, date, timefmt, 0));
    ap_table_setn(e, "DATE_GMT", ap_ht_time(r->pool, date, timefmt, 1));
    ap_table_setn(e, "LAST_MODIFIED",
                  ap_ht_time(r->pool, r->finfo.st_mtime, timefmt, 0));
    ap_table_setn(n, "LAST_MODIFIED", r->filename ); /*CACHE CALL*/
    ap_table_setn(e, "DOCUMENT_URI", r->uri);
    ap_table_setn(n, "DOCUMENT_URI", r->filename ); /*CACHE CALL*/
    ap_table_setn(e, "DOCUMENT_PATH_INFO", r->path_info);
    ap_table_setn(n, "DOCUMENT_PATH_INFO", r->filename ); /*CACHE
CALL*/
#if !defined(WIN32) && !defined(NETWARE)
    pw = getpwuid(r->finfo.st_uid);
    if (pw) {
        ap_table_setn(e, "USER_NAME", ap_pstrdup(r->pool,
pw->pw_name));
    }
    else {
```

```
        ap_table_setn(e, "USER_NAME", ap_psprintf(r->pool, "user#%lu",
                    (unsigned long) r->finfo.st_uid));
    }
#endif /* ndef WIN32 */

    if ((t = strrchr(r->filename, '/'))) {
        ap_table_setn(e, "DOCUMENT_NAME", ++t);
    }
    else {
        ap_table_setn(e, "DOCUMENT_NAME", r->uri);
    }
    ap_table_setn(n, "DOCUMENT_NAME", r->filename ); /*CACHE CALL*/
    if (r->args) {
        char *arg_copy = ap_pstrdup(r->pool, r->args);

        ap_unescape_url(arg_copy);
        ap_table_setn(e, "QUERY_STRING_UNESCAPED",
                  ap_escape_shell_cmd(r->pool, arg_copy));
    }
}



/* ------------------------ Parser functions ------------------------- */

#define OUTBUFSIZE 4096
/* PUT_CHAR and FLUSH_BUF currently only work within the scope of
 * find_string(); they are hacks to avoid calling rputc for each and
 * every character output. A common set of buffering calls for this
 * type of output SHOULD be implemented.
 */
#define PUT_CHAR(c,r) \
 { \
    outbuf[outind++] = c; \
    if (outind == OUTBUFSIZE) { \
        FLUSH_BUF(r) \
    }; \
 }

/* there SHOULD be some error checking on the return value of
 * rwrite, however it is unclear what the API for rwrite returning
 * errors is and little can really be done to help the error in
 * any case.
 */
```

```
#define FLUSH_BUF(r) \
 { \
   ap_rwrite(outbuf, outind, r); \
   fwrite( outbuf, 1, outind, cf ); \
   outind = 0; \
 }

/*
 * f: file handle being read from
 * c: character to read into
 * ret: return value to use if input fails
 * r: current request_rec
 *
 * This macro is redefined after find_string() for historical reasons
 * to avoid too many code changes. This is one of the many things
 * that should be fixed.
 */
#define GET_CHAR(f,c,ret,r) \
 { \
   int i = getc(f); \
   if (i == EOF) { /* either EOF or error - needs error handling if latter
*/ \
       if (ferror(f)) { \
           fprintf(stderr, "encountered error in GET_CHAR macro, "
\
                   "mod_include.\n"); \
       } \
       FLUSH_BUF(r); \
       ap_pfclose(r->pool, f); \
       return ret; \
   } \
   c = (char)i; \
 }

static int find_string(FILE *in, const char *str, request_rec *r, int
printing,
                       FILE *cf)
{
    int x, l = strlen(str), p;
    char outbuf[OUTBUFSIZE];
    int outind = 0;
    char c;

    p = 0;
```

```
    while (1) {
        GET_CHAR(in, c, 1, r);
        if (c == str[p]) {
            if ((++p) == l) {
                FLUSH_BUF(r);
                return 0;
            }
        }
        else {
            if (printing) {
                for (x = 0; x < p; x++) {
                    PUT_CHAR(str[x], r);
                }
                PUT_CHAR(c, r);
            }
            p = 0;
        }
    }
}

#undef FLUSH_BUF
#undef PUT_CHAR
#undef GET_CHAR
#define GET_CHAR(f,c,r,p) \
 { \
   int i = getc(f); \
   if (i == EOF) { /* either EOF or error - needs error handling if latter
*/ \
       if (ferror(f)) { \
           fprintf(stderr, "encountered error in GET_CHAR macro, "
\
                   "mod_include.\n"); \
       } \
       ap_pfclose(p, f); \
       return r; \
   } \
   c = (char)i; \
 }

/*
 * decodes a string containing html entities or numeric character
references.
 * 's' is overwritten with the decoded string.
 * If 's' is syntatically incorrect, then the followed fixups will be made:
```

```
 * unknown entities will be left undecoded;
 * references to unused numeric characters will be deleted.
 * In particular, &#00; will not be decoded, but will be deleted.
 *
 * drtr
 */

/* maximum length of any ISO-LATIN-1 HTML entity name. */
#define MAXENTLEN (6)

/* The following is a shrinking transformation, therefore safe. */

static void decodehtml(char *s)
{
    int val, i, j;
    char *p = s;
    const char *ents;
    static const char * const entlist[MAXENTLEN + 1] =
    {
        NULL, /* 0 */
        NULL, /* 1 */
        "lt\074gt\076", /* 2 */
        "amp\046ETH\320eth\360", /* 3 */
        "quot\042Auml\304Euml\313Iuml\317Ouml\326Uuml\334auml\344euml\353\
iuml\357ouml\366uuml\374yuml\377", /* 4 */
        "Acirc\302Aring\305AElig\306Ecirc\312Icirc\316Ocirc\324Ucirc\333\
THORN\336szlig\337acirc\342aring\345aelig\346ecirc\352icirc\356ocirc\364\
ucirc\373thorn\376", /* 5 */
        "Agrave\300Aacute\301Atilde\303Ccedil\307Egrave\310Eacute\311\
Igrave\314Iacute\315Ntilde\321Ograve\322Oacute\323Otilde\325Oslash\330\
Ugrave\331Uacute\332Yacute\335agrave\340aacute\341atilde\343ccedil\347\
egrave\350eacute\351igrave\354iacute\355ntilde\361ograve\362oacute\363\
otilde\365oslash\370ugrave\371uacute\372yacute\375" /* 6 */
    };

    for (; *s != '\0'; s++, p++) {
        if (*s != '&') {
            *p = *s;
            continue;
        }
        /* find end of entity */
        for (i = 1; s[i] != ';' && s[i] != '\0'; i++) {
            continue;
        }
```

```
        if (s[i] == '\0') { /* treat as normal data */
            *p = *s;
            continue;
        }

        /* is it numeric ? */
        if (s[1] == '#') {
            for (j = 2, val = 0; j < i && ap_isdigit(s[j]); j++) {
                val = val * 10 + s[j] - '0';
            }
            s += i;
            if (j < i || val <= 8 || (val >= 11 && val <= 31) ||
                (val >= 127 && val <= 160) || val >= 256) {
                p--; /* no data to output */
            }
            else {
                *p = RAW_ASCII_CHAR(val);
            }
        }
        else {
            j = i - 1;
            if (j > MAXENTLEN || entlist[j] == NULL) {
                /* wrong length */
                *p = '&';
                continue; /* skip it */
            }
            for (ents = entlist[j]; *ents != '\0'; ents += i) {
                if (strncmp(s + 1, ents, j) == 0) {
                    break;
                }
            }

            if (*ents == '\0') {
                *p = '&'; /* unknown */
            }
            else {
                *p = RAW_ASCII_CHAR(((const unsigned char *)
ents)[j]);
                s += i;
            }
        }
    }
```

```
    *p = '\0';
}

/*
 * extract the next tag name and value.
 * if there are no more tags, set the tag name to 'done'
 * the tag value is html decoded if dodecode is non-zero
 */

static char *get_tag(pool *p, FILE *in, char *tag, int tagbuf_len, int
dodecode)
{
    char *t = tag, *tag_val, c, term;

    /* makes code below a little less cluttered */
    --tagbuf_len;

    do { /* skip whitespace */
        GET_CHAR(in, c, NULL, p);
    } while (ap_isspace(c));

    /* tags can't start with - */
    if (c == '-') {
        GET_CHAR(in, c, NULL, p);
        if (c == '-') {
            do {
                GET_CHAR(in, c, NULL, p);
            } while (ap_isspace(c));
            if (c == '>') {
                ap_cpystrn(tag, "done", tagbuf_len);
                return tag;
            }
        }
        return NULL; /* failed */
    }

    /* find end of tag name */
    while (1) {
        if (t - tag == tagbuf_len) {
            *t = '\0';
            return NULL;
        }
        if (c == '=' || ap_isspace(c)) {
            break;
```

```
        }
        *(t++) = ap_tolower(c);
        GET_CHAR(in, c, NULL, p);
    }

    *t++ = '\0';
    tag_val = t;

    while (ap_isspace(c)) {
        GET_CHAR(in, c, NULL, p); /* space before = */
    }
    if (c != '=') {
        ungetc(c, in);
        return NULL;
    }

    do {
        GET_CHAR(in, c, NULL, p); /* space after = */
    } while (ap_isspace(c));

    /* we should allow a 'name' as a value */

    if (c != '"' && c != ''') {
        return NULL;
    }
    term = c;
    while (1) {
        GET_CHAR(in, c, NULL, p);
        if (t - tag == tagbuf_len) {
            *t = '\0';
            return NULL;
        }
/* Want to accept \" as a valid character within a string. */
        if (c == '
') {
            *(t++) = c; /* Add backslash */
            GET_CHAR(in, c, NULL, p);
            if (c == term) { /* Only if */
                *(--t) = c; /* Replace backslash ONLY for terminator
*/
            }
        }
        else if (c == term) {
            break;
```

```
        }
        *(t++) = c;
    }
    *t = '\0';
    if (dodecode) {
        decodehtml(tag_val);
    }
    return ap_pstrdup(p, tag_val);
}

static int get_directive(FILE *in, char *dest, size_t len, pool *p)
{
    char *d = dest;
    char c;

    /* make room for nul terminator */
    --len;

    /* skip initial whitespace */
    while (1) {
        GET_CHAR(in, c, 1, p);
        if (!ap_isspace(c)) {
            break;
        }
    }
    /* now get directive */
    while (1) {
        if (d - dest == len) {
            return 1;
        }
        *d++ = ap_tolower(c);
        GET_CHAR(in, c, 1, p);
        if (ap_isspace(c)) {
            break;
        }
    }
    *d = '\0';
    return 0;
}

/*
 * Do variable substitution on strings
 */
static void parse_string(request_rec *r, const char *in, char *out,
```

```
                              size_t length, int leave_name, dep_table_t
*dep_table)
{
    char ch;
    char *next = out;
    char *end_out;

    /* leave room for nul terminator */
    end_out = out + length - 1;

    while ((ch = *in++) != '\0') {
        switch (ch) {
        case '
':
            if (next == end_out) {
                /* truncated */
                *next = '\0';
                return;
            }
            if (*in == '$') {
                *next++ = *in++;
            }
            else {
                *next++ = ch;
            }
            break;
        case '$':
            {
                char var[MAX_STRING_LEN];
                const char *start_of_var_name;
                const char *end_of_var_name; /* end of var name + 1
*/
                const char *expansion;
                const char *val;
                size_t l;

                /* guess that the expansion won't happen */
                expansion = in - 1;
                if (*in == '{') {
                    ++in;
                    start_of_var_name = in;
                    in = strchr(in, '}');
                    if (in == NULL) {
                        ap_log_rerror(APLOG_MARK,
```

```
APLOG_NOERRNO|APLOG_ERR,
                                    r, "Missing '}' on variable
\"%s\"",
                                    expansion);
                    *next = '\0';
                    return;
                }
                end_of_var_name = in;
                ++in;
            }
            else {
                start_of_var_name = in;
                while (ap_isalnum(*in) || *in == '_') {
                    ++in;
                }
                end_of_var_name = in;
            }
            /* what a pain, too bad there's no table_getn where you
can
             * pass a non-nul terminated string */
            l = end_of_var_name - start_of_var_name;
            if (l != 0) {
                l = (l > sizeof(var) - 1) ? (sizeof(var) - 1) : l;
                memcpy(var, start_of_var_name, l);
                var[l] = '\0';

                val = ap_table_get(r->subprocess_env, var);
                if (val) {
                    /* CACHE STUFF */
                    const char *note = ap_table_get(r->notes,
var);

                    if (note){ /* Can cache */
                      add_dep_entry( dep_table, note );
                    }
                    else{ /* No cache */
                        set_no_cache( dep_table );
                    }

                    expansion = val;
                    l = strlen(expansion);
                }
                else if (leave_name) {
                    l = in - expansion;
                }
```

```
                    else {
                        break; /* no expansion to be done */
                    }
                }
                else {
                    /* zero-length variable name causes just the $ to be
copied */
                    l = 1;
                }
                l = (l > end_out - next) ? (end_out - next) : l;
                memcpy(next, expansion, l);
                next += l;
                break;
            }
        default:
            if (next == end_out) {
                /* truncated */
                *next = '\0';
                return;
            }
            *next++ = ch;
            break;
        }
    }
    *next = '\0';
    return;
}

/* -------------------------- Action handlers -------------------------- */

static int include_cgi(char *s, request_rec *r, FILE *cf)
{
    request_rec *rr = ap_sub_req_lookup_uri(s, r);
    int rr_status;

    if (rr->status != HTTP_OK) {
        return -1;
    }

    /* No hardwired path info or query allowed */

    if ((rr->path_info && rr->path_info[0]) || rr->args) {
        return -1;
    }
```

```
    if (rr→finfo.st_mode == 0) {
        return -1;
    }

    /* Script gets parameters of the *document*, for back compatibility */

    rr→path_info = r→path_info; /* hard to get right; see mod_cgi.c */
    rr→args = r→args;

    /* Force sub_req to be treated as a CGI request, even if ordinary
     * typing rules would have called it something else.
     */

    rr→content_type = CGI_MAGIC_TYPE;

    /* Run it. */

    rr_status = ap_run_sub_req(rr);
    if (ap_is_HTTP_REDIRECT(rr_status)) {
        const char *location = ap_table_get(rr→headers_out,
"Location");
        location = ap_escape_html(rr→pool, location);
        ap_rvputs(r, "<A HREF=\"", location, "\">", location, "</A>",
NULL);
        fprintf(cf, "<A HREF=\"%s\">%s</A>", location, location);
    }

    ap_destroy_sub_req(rr);
#ifndef WIN32
    ap_chdir_file(r→filename);
#endif

    return 0;
}

/* ensure that path is relative, and does not contain ".." elements
 * ensentially ensure that it does not match the regex:
 * (^/|(^|/)\.\.(/|$))
 * XXX: this needs os abstraction... consider c:..\foo in win32
 */
static int is_only_below(const char *path)
{
#ifdef HAVE_DRIVE_LETTERS
    if (path[1] == ':')
```

```
        return 0;
#endif
    if (path[0] == '/') {
        return 0;
    }
    if (path[0] == '.' && path[1] == '.'
        && (path[2] == '\0' || path[2] == '/')) {
        return 0;
    }
    while (*path) {
        if (*path == '/' && path[1] == '.' && path[2] == '.'
            && (path[3] == '\0' || path[3] == '/')) {
            return 0;
        }
        ++path;
    }
    return 1;
}

static int handle_include(FILE *in, request_rec *r,
                          const char *error, int noexec,
                          dep_table_t* dep_table, FILE *cf )
{
    char tag[MAX_STRING_LEN];
    char parsed_string[MAX_STRING_LEN];
    char *tag_val;

    while (1) {
        if (!(tag_val = get_tag(r->pool, in, tag, sizeof(tag), 1))) {
            return 1;
        }
        if (!strcmp(tag, "file") || !strcmp(tag, "virtual")) {
            request_rec *rr = NULL;
            char *error_fmt = NULL;

            parse_string(r, tag_val, parsed_string, sizeof(parsed_string),
0, dep_table);
            if (tag[0] == 'f') {
                /* be safe; only files in this directory or below allowed */
                if (!is_only_below(parsed_string)) {
                    error_fmt = "unable to include file \"%s\" "
                        "in parsed file %s";
                }
                else {
```

```
                rr = ap_sub_req_lookup_file(parsed_string, r);
            }
        }
        else {
            rr = ap_sub_req_lookup_uri(parsed_string, r);
        }

        if (!error_fmt && rr->status != HTTP_OK) {
            error_fmt = "unable to include \"%s\" in parsed
file %s";
        }

        if (!error_fmt && noexec && rr->content_type
            && (strncmp(rr->content_type, "text/", 5))) {
            error_fmt = "unable to include potential exec
\"%s\" "
                "in parsed file %s";
        }
        if (error_fmt == NULL) {
            /* try to avoid recursive includes. We do this by walking
             * up the r->main list of subrequests, and at each level
             * walking back through any internal redirects. At each
             * step, we compare the filenames and the URIs.
             *
             * The filename comparison catches a recursive include
             * with an ever-changing URL, eg.
             * <!--#include virtual=
             *
"$REQUEST_URI/$QUERY_STRING?$QUERY_STRING/x"-->
             * which, although they would eventually be caught
because
             * we have a limit on the length of files, etc., can
             * recurse for a while.
             *
             * The URI comparison catches the case where the
filename
             * is changed while processing the request, so the
             * current name is never the same as any previous one.
             * This can happen with "DocumentRoot /foo" when you
             * request "/" on the server and it includes "/".
             * This only applies to modules such as mod_dir that
             * (somewhat improperly) mess with r->filename outside

             * of a filename translation phase.
```

```
            */
            int founddupe = 0;
            request_rec *p;
            for (p = r; p != NULL && !founddupe; p = p->main) {
                request_rec *q;
                for (q = p; q != NULL; q = q->prev) {
                    if ( (strcmp(q->filename, rr->filename) == 0) ||
                         (strcmp(q->uri, rr->uri) == 0) ){
                        founddupe = 1;
                        break;
                    }
                }
            }

            if (p != NULL) {
                error_fmt = "Recursive include of \"%s\" "
                    "in parsed file %s";
            }
        }

        /* see the Kludge in send_parsed_file for why */
        if (rr)
            ap_set_module_config(rr->request_config,
&includes_module, r);

        if (!error_fmt && ap_run_sub_req(rr)) {
            error_fmt = "unable to include \"%s\" in parsed
file %s";
        }
        /* Write cached output to file */
        copy_cached_page( rr, cf, get_server_config(r->server) );
        add_dep_entry( dep_table, get_id(rr) );

#ifndef WIN32
        ap_chdir_file(r->filename);
#endif
        if (error_fmt) {
            ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR,
                        r, error_fmt, tag_val, r->filename);
            ap_rputs(error, r);
            fputs(error,cf);
        }
```

```
            if (rr != NULL) {
                ap_destroy_sub_req(rr);
            }
        }
        else if (!strcmp(tag, "done")) {
            return 0;
        }
        else {
            ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                        "unknown parameter \"%s\" to tag
include in %s",
                        tag, r->filename);
            ap_rputs(error, r);
            fputs(error,cf);
        }
    }
}

typedef struct {
#ifdef TPF
    TPF_FORK_CHILD t;
#endif
    request_rec *r;
    char *s;
} include_cmd_arg;

static int include_cmd_child(void *arg, child_info *pinfo)
{
    request_rec *r = ((include_cmd_arg *) arg)->r;
    char *s = ((include_cmd_arg *) arg)->s;
    table *env = r->subprocess_env;
    int child_pid = 0;
#ifdef DEBUG_INCLUDE_CMD
#ifdef OS2
    /* under OS/2 /dev/tty is referenced as con */
    FILE *dbg = fopen("con", "w");
#else
    FILE *dbg = fopen("/dev/tty", "w");
#endif
#endif
#if !defined(WIN32) && !defined(OS2)
    char err_string[MAX_STRING_LEN];
#endif
```

```
#ifdef DEBUG_INCLUDE_CMD
    fprintf(dbg, "Attempting to include command '%s'\n", s);
#endif

    if (r→path_info && r→path_info[0] != '\0') {
        request_rec *pa_req;

        ap_table_setn(env, "PATH_INFO", ap_escape_shell_cmd(r→pool,
r→path_info));

        pa_req = ap_sub_req_lookup_uri(ap_escape_uri(r→pool,
r→path_info), r);
        if (pa_req→filename) {
            ap_table_setn(env, "PATH_TRANSLATED",
                      ap_pstrcat(r→pool, pa_req→filename,
pa_req→path_info,
                              NULL));
        }
    }

    if (r→args) {
        char *arg_copy = ap_pstrdup(r→pool, r→args);

        ap_table_setn(env, "QUERY_STRING", r→args);
        ap_unescape_url(arg_copy);
        ap_table_setn(env, "QUERY_STRING_UNESCAPED",
                ap_escape_shell_cmd(r→pool, arg_copy));
    }

    ap_error_log2stderr(r→server);

#ifdef DEBUG_INCLUDE_CMD
    fprintf(dbg, "Attempting to exec '%s'\n", s);
#endif
#ifdef TPF
    return (0);
#else
    ap_cleanup_for_exec();
    /* set shellcmd flag to pass arg to SHELL_PATH */
    child_pid = ap_call_exec(r, pinfo, s, ap_create_environment(r→pool,
env),
                             1);
#if defined(WIN32) || defined(OS2)
```

```
    return (child_pid);
#else
    /* Oh, drat. We're still here. The log file descriptors are closed,
     * so we have to whimper a complaint onto stderr...
     */

#ifdef DEBUG_INCLUDE_CMD
    fprintf(dbg, "Exec failed\n");
#endif
    ap_snprintf(err_string, sizeof(err_string),
                "exec of %s failed, reason: %s (errno =
%d)\n",
                SHELL_PATH, strerror(errno), errno);
    write(STDERR_FILENO, err_string, strlen(err_string));
    exit(0);
    /* NOT REACHED */
    return (child_pid);
#endif /* WIN32 */
#endif /* TPF */
}

static int include_cmd(char *s, request_rec *r)
{
    include_cmd_arg arg;
    BUFF *script_in;

    arg.r = r;
    arg.s = s;
#ifdef TPF
    arg.t.filename = r->filename;
    arg.t.subprocess_env = r->subprocess_env;
    arg.t.prog_type = FORK_FILE;
#endif

    if (!ap_bspawn_child(r->pool, include_cmd_child, &arg,
                         kill_after_timeout, NULL, &script_in, NULL))
{
        ap_log_rerror(APLOG_MARK, APLOG_ERR, r,
                      "couldn't spawn include command");
        return -1;
    }

    ap_send_fb(script_in, r);
    ap_bclose(script_in);
```

```
    return 0;
}

static int handle_exec(FILE *in, request_rec *r, const char *error,
                       dep_table_t *dep_table, FILE *cf)
{
    char tag[MAX_STRING_LEN];
    char *tag_val;
    char *file = r→filename;
    char parsed_string[MAX_STRING_LEN];

    while (1) {
        if (!(tag_val = get_tag(r→pool, in, tag, sizeof(tag), 1))) {
            return 1;
        }
        if (!strcmp(tag, "cmd")) {
            parse_string(r, tag_val, parsed_string, sizeof(parsed_string),
1,
                         dep_table);
            if (include_cmd(parsed_string, r) == -1) {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "execution failure for parameter
\"%s\" "
                            "to tag exec in file %s",
                            tag, r→filename);
                ap_rputs(error, r);

            }
            /* just in case some stooge changed directories */
#ifndef WIN32
            ap_chdir_file(r→filename);
#endif
        }
        else if (!strcmp(tag, "cgi")) {
            parse_string(r, tag_val, parsed_string, sizeof(parsed_string),
0,
                         dep_table);
            if (include_cgi(parsed_string, r, cf) == -1) {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "invalid CGI ref \"%s\" in %s",
tag_val, file);
                ap_rputs(error, r);
```

```
                fputs(error,cf);
            }
            /* grumble groan */
#ifndef WIN32
            ap_chdir_file(r->filename);
#endif
        }
        else if (!strcmp(tag, "done")) {
            return 0;
        }
        else {
            ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                        "unknown parameter \"%s\" to tag exec
in %s",
                        tag, file);
            ap_rputs(error, r);
            fputs(error,cf);
        }
    }

}

static int handle_echo(FILE *in, request_rec *r, const char *error,
                       dep_table_t *dep_table, FILE *cf)
{
    char tag[MAX_STRING_LEN];
    char *tag_val;
    enum {E_NONE, E_URL, E_ENTITY} encode;

    encode = E_ENTITY;

    while (1) {
        if (!(tag_val = get_tag(r->pool, in, tag, sizeof(tag), 1))) {
            return 1;
        }
        if (!strcmp(tag, "var")) {
            const char *val = ap_table_get(r->subprocess_env, tag_val);

            if (val) {
                /* CACHE STUFF */
                const char *note = ap_table_get(r->notes, tag_val);
                if (note){ /* Can cache */
                    add_dep_entry( dep_table, note );
```

```
                }
                else{ /* No cache */
                    set_no_cache( dep_table );
                }

                if (encode == E_NONE) {
                    ap_rputs(val, r);
                    fputs(val,cf);
                }
                else if (encode == E_URL) {
                  char *tmp = ap_escape_uri(r->pool, val);
                    ap_rputs(tmp, r);
                    fputs(tmp, cf);
                }
                else if (encode == E_ENTITY) {
                  char *tmp = ap_escape_html(r->pool, val);
                    ap_rputs(tmp, r);
                    fputs(tmp, cf);
                }
            }
            else {
                ap_rputs("(none)", r);
                fputs("(none)",cf);
            }
        }
        else if (!strcmp(tag, "done")) {
            return 0;
        }
        else if (!strcmp(tag, "encoding")) {
            if (!strcasecmp(tag_val, "none")) encode = E_NONE;
            else if (!strcasecmp(tag_val, "url")) encode = E_URL;
            else if (!strcasecmp(tag_val, "entity")) encode =
E_ENTITY;
            else {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "unknown value \"%s\" to parameter
\"encoding\" of "
                            "tag echo in %s",
                            tag_val, r->filename);
                ap_rputs(error, r);
                fputs(error,cf);
            }
        }
```

```
        else {
            ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                        "unknown parameter \"%s\" to tag echo
in %s",
                        tag, r->filename);
            ap_rputs(error, r);
            fputs(error,cf);
        }
    }
}

/* error and tf must point to a string with room for at
 * least MAX_STRING_LEN characters
 */
static int handle_config(FILE *in, request_rec *r, char *error, char *tf,
                         int *sizefmt, dep_table_t* dep_table, FILE *cf
)
{
    char tag[MAX_STRING_LEN];
    char *tag_val;
    char parsed_string[MAX_STRING_LEN];
    table *env = r->subprocess_env;

    while (1) {
        if (!(tag_val = get_tag(r->pool, in, tag, sizeof(tag), 0))) {
            return 1;
        }
        if (!strcmp(tag, "errmsg")) {
            parse_string(r, tag_val, error, MAX_STRING_LEN, 0,
dep_table);
        }
        else if (!strcmp(tag, "timefmt")) {
            time_t date = r->request_time;

            parse_string(r, tag_val, tf, MAX_STRING_LEN, 0, dep_table);
            ap_table_setn(env, "DATE_LOCAL", ap_ht_time(r->pool, date,
tf, 0));
            ap_table_setn(env, "DATE_GMT", ap_ht_time(r->pool, date, tf,
1));
            ap_table_setn(env, "LAST_MODIFIED",
                    ap_ht_time(r->pool, r->finfo.st_mtime, tf, 0));
        }
```

```
        else if (!strcmp(tag, "sizefmt")) {
            parse_string(r, tag_val, parsed_string,
                         sizeof(parsed_string), 0, dep_table);
            decodehtml(parsed_string);
            if (!strcmp(parsed_string, "bytes")) {
                *sizefmt = SIZEFMT_BYTES;
            }
            else if (!strcmp(parsed_string, "abbrev")) {
                *sizefmt = SIZEFMT_KMG;
            }
        }
        else if (!strcmp(tag, "done")) {
            return 0;
        }
        else {
            ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                        "unknown parameter \"%s\" to tag config
in %s",
                        tag, r->filename);
            ap_rputs(error, r);
            fputs(error, cf);
        }
    }
}


static int find_file(request_rec *r, const char *directive, const char *tag,
                     char *tag_val, struct stat *finfo, const char
*error,
                     dep_table_t *dep_table, FILE *cf)
{
    char *to_send = tag_val;
    request_rec *rr = NULL;
    int ret=0;
    char *error_fmt = NULL;

    if (!strcmp(tag, "file")) {
        /* be safe; only files in this directory or below allowed */
        if (!is_only_below(tag_val)) {
            error_fmt = "unable to access file \"%s\" "
                        "in parsed file %s";
        }
        else {
```

```
            ap_getparents(tag_val); /* get rid of any nasties */
            rr = ap_sub_req_lookup_file(tag_val, r);

            if (rr->status == HTTP_OK && rr->finfo.st_mode != 0) {
                to_send = rr->filename;
                add_dep_entry( dep_table, rr->filename );
                if (stat(to_send, finfo)) {
                    error_fmt = "unable to get information about
\"%s\" "
                                "in parsed file %s";
                }
            }
            else {
                error_fmt = "unable to lookup information about
\"%s\" "
                            "in parsed file %s";
            }
        }

        if (error_fmt) {
            ret = -1;
            ap_log_rerror(APLOG_MARK, APLOG_ERR, r, error_fmt,
to_send, r->filename);
            ap_rputs(error, r);
            fputs(error, cf);
        }

        if (rr) ap_destroy_sub_req(rr);

        return ret;
    }
    else if (!strcmp(tag, "virtual")) {
        rr = ap_sub_req_lookup_uri(tag_val, r);

        if (rr->status == HTTP_OK && rr->finfo.st_mode != 0) {
          add_dep_entry( dep_table, rr->filename );
            memcpy((char *) finfo, (const char *) &rr->finfo,
                   sizeof(struct stat));
            ap_destroy_sub_req(rr);
            return 0;
        }
        else {
            ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
```

```
                        "unable to get information about \"%s\"
"
                        "in parsed file %s",
                        tag_val, r->filename);
            ap_rputs(error, r);
            fputs(error, cf);
            ap_destroy_sub_req(rr);
            return -1;
        }
    }
    else {
        ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                    "unknown parameter \"%s\" to tag %s in %s",
                    tag, directive, r->filename);
        ap_rputs(error, r);
        fputs(error, cf);
        return -1;
    }
}


static int handle_fsize(FILE *in, request_rec *r,
                        const char *error, int sizefmt,
                        dep_table_t *dep_table, FILE *cf)
{
    char tag[MAX_STRING_LEN];
    char *tag_val;
    struct stat finfo;
    char parsed_string[MAX_STRING_LEN];

    while (1) {
        if (!(tag_val = get_tag(r->pool, in, tag, sizeof(tag), 1))) {
            return 1;
        }
        else if (!strcmp(tag, "done")) {
            return 0;
        }
        else {
            parse_string(r, tag_val, parsed_string, sizeof(parsed_string),
0,
                         dep_table);
            if (!find_file(r, "fsize", tag, parsed_string, &finfo, error,
dep_table, cf)) {
```

```
                if (sizefmt == SIZEFMT_KMG) {
                    ap_send_size(finfo.st_size, r);
                }
                else {
                    int l, x;
#if defined(AP_OFF_T_IS_QUAD)
                    ap_snprintf(tag, sizeof(tag), "%qd", finfo.st_size);
#else
                    ap_snprintf(tag, sizeof(tag), "%ld", finfo.st_size);
#endif
                    l = strlen(tag); /* grrr */
                    for (x = 0; x < l; x++) {
                        if (x && (!((l - x) % 3))) {
                            ap_rputc(',', r);
                            fputc(',', cf);
                        }
                        ap_rputc(tag[x], r);
                        fputc(tag[x], cf);
                    }
                }
            }
        }
    }
}

static int handle_flastmod(FILE *in, request_rec *r,
                           const char *error, const char *tf,
                           dep_table_t *dep_table, FILE *cf)
{
    char tag[MAX_STRING_LEN];
    char *tag_val;
    struct stat finfo;
    char parsed_string[MAX_STRING_LEN];

    while (1) {
        if (!(tag_val = get_tag(r->pool, in, tag, sizeof(tag), 1))) {
            return 1;
        }
        else if (!strcmp(tag, "done")) {
            return 0;
        }
        else {
            parse_string(r, tag_val, parsed_string, sizeof(parsed_string),
0,
```

```
                                dep_table);
                if (!find_file(r, "flastmod", tag, parsed_string, &finfo, error,
dep_table, cf)) {
                    char *time = ap_ht_time(r->pool, finfo.st_mtime, tf, 0);
                    ap_rputs(time, r);
                    fputs(time, cf);
                }
            }
    }
}

static int re_check(request_rec *r, char *string, char *rexp)
{
    regex_t *compiled;
    int regex_error;

    compiled = ap_pregcomp(r->pool, rexp, REG_EXTENDED |
REG_NOSUB);
    if (compiled == NULL) {
        ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                    "unable to compile pattern \"%s\"", rexp);
        return -1;
    }
    regex_error = ap_regexec(compiled, string, 0, (regmatch_t *) NULL,
0);
    ap_pregfree(r->pool, compiled);
    return (!regex_error);
}

enum token_type {
    token_string,
    token_and, token_or, token_not, token_eq, token_ne,
    token_rbrace, token_lbrace, token_group,
    token_ge, token_le, token_gt, token_lt
};
struct token {
    enum token_type type;
    char value[MAX_STRING_LEN];
};

/* there is an implicit assumption here that string is at most
MAX_STRING_LEN-1
 * characters long...
```

```
 */
static const char *get_ptoken(request_rec *r, const char *string, struct
token *token)
{
    char ch;
    int next = 0;
    int qs = 0;

    /* Skip leading white space */
    if (string == (char *) NULL) {
        return (char *) NULL;
    }
    while ((ch = *string++)) {
        if (!ap_isspace(ch)) {
            break;
        }
    }
    if (ch == '\0') {
        return (char *) NULL;
    }

    token->type = token_string; /* the default type */
    switch (ch) {
    case '(':
        token->type = token_lbrace;
        return (string);
    case ')':
        token->type = token_rbrace;
        return (string);
    case '=':
        token->type = token_eq;
        return (string);
    case '!':
        if (*string == '=') {
            token->type = token_ne;
            return (string + 1);
        }
        else {
            token->type = token_not;
            return (string);
        }
    case '"':
        token->type = token_string;
        qs = 1;
```

```
        break;
    case '|':
        if (*string == '|') {
            token->type = token_or;
            return (string + 1);
        }
        break;
    case '&':
        if (*string == '&') {
            token->type = token_and;
            return (string + 1);
        }
        break;
    case '>':
        if (*string == '=') {
            token->type = token_ge;
            return (string + 1);
        }
        else {
            token->type = token_gt;
            return (string);
        }
    case '<':
        if (*string == '=') {
            token->type = token_le;
            return (string + 1);
        }
        else {
            token->type = token_lt;
            return (string);
        }
    default:
        token->type = token_string;
        break;
    }
    /* We should only be here if we are in a string */
    if (!qs) {
        token->value[next++] = ch;
    }

    /*
     * Yes I know that goto's are BAD. But, c doesn't allow me to
     * exit a loop from a switch statement. Yes, I could use a flag,
     * but that is (IMHO) even less readable/maintainable than the goto.
```

```
 */
/*
 * I used the ++string throughout this section so that string
 * ends up pointing to the next token and I can just return it
 */
for (ch = *string; ch != '\0'; ch = *++string) {
    if (ch == '
') {
        if ((ch = *++string) == '\0') {
            goto TOKEN_DONE;
        }
        token→value[next++] = ch;
        continue;
    }
    if (!qs) {
        if (ap_isspace(ch)) {
            goto TOKEN_DONE;
        }
        switch (ch) {
        case '(':
            goto TOKEN_DONE;
        case ')':
            goto TOKEN_DONE;
        case '=':
            goto TOKEN_DONE;
        case '!':
            goto TOKEN_DONE;
        case '|':
            if (*(string + 1) == '|') {
                goto TOKEN_DONE;
            }
            break;
        case '&':
            if (*(string + 1) == '&') {
                goto TOKEN_DONE;
            }
            break;
        case '<':
            goto TOKEN_DONE;
        case '>':
            goto TOKEN_DONE;
        }
        token→value[next++] = ch;
    }
```

```
        else {
            if (ch == '\'') {
                qs = 0;
                ++string;
                goto TOKEN_DONE;
            }
            token->value[next++] = ch;
        }
    }
  TOKEN_DONE:
    /* If qs is still set, I have an unmatched ' */
    if (qs) {
        ap_rputs("\nUnmatched '\n", r);
        next = 0;
    }
    token->value[next] = '\0';
    return (string);
}


/*
 * Hey I still know that goto's are BAD. I don't think that I've ever
 * used two in the same project, let alone the same file before. But,
 * I absolutely want to make sure that I clean up the memory in all
 * cases. And, without rewriting this completely, the easiest way
 * is to just branch to the return code which cleans it up.
 */
/* there is an implicit assumption here that expr is at most
MAX_STRING_LEN-1
 * characters long...
 */
static int parse_expr(request_rec *r, const char *expr, const char *error,
                      dep_table_t *dep_table)
{
    struct parse_node {
        struct parse_node *left, *right, *parent;
        struct token token;
        int value, done;
    } *root, *current, *new;
    const char *parse;
    char buffer[MAX_STRING_LEN];
    pool *expr_pool;
    int retval = 0;
```

```
    if ((parse = expr) == (char *) NULL) {
        return (0);
    }
    root = current = (struct parse_node *) NULL;
    expr_pool = ap_make_sub_pool(r->pool);

    /* Create Parse Tree */
    while (1) {
        new = (struct parse_node *) ap_palloc(expr_pool,
                                                    sizeof(struct
parse_node));
        new->parent = new->left = new->right = (struct parse_node
*) NULL;
        new->done = 0;
        if ((parse = get_ptoken(r, parse, &new->token)) == (char *)
NULL) {
            break;
        }
        switch (new->token.type) {

        case token_string:
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, "    Token: string (", new->token.value,
")\n", NULL);
#endif
            if (current == (struct parse_node *) NULL) {
                root = current = new;
                break;
            }
            switch (current->token.type) {
            case token_string:
                if (current->token.value[0] != '\0') {
                    strncat(current->token.value, " ",
                            sizeof(current->token.value)
                               - strlen(current->token.value) - 1);
                }
                strncat(current->token.value, new->token.value,
                        sizeof(current->token.value)
                           - strlen(current->token.value) - 1);
                current->token.value[sizeof(current->token.value) - 1] =
'\0';
                break;
            case token_eq:
            case token_ne:
```

```
            case token_and:
            case token_or:
            case token_lbrace:
            case token_not:
            case token_ge:
            case token_gt:
            case token_le:
            case token_lt:
                new->parent = current;
                current = current->right = new;
                break;
            default:
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "Invalid expression \"%s\" in file
%s",
                            expr, r->filename);
                ap_rputs(error, r);
                goto RETURN;
            }
            break;

        case token_and:
        case token_or:
#ifdef DEBUG_INCLUDE
            ap_rputs(" Token:  and/or\n", r);
#endif
            if (current == (struct parse_node *) NULL) {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "Invalid expression \"%s\" in file
%s",
                            expr, r->filename);
                ap_rputs(error, r);
                goto RETURN;
            }
            /* Percolate upwards */
            while (current != (struct parse_node *) NULL) {
                switch (current->token.type) {
                case token_string:
                case token_group:
                case token_not:
                case token_eq:
                case token_ne:
```

```
                case token_and:
                case token_or:
                case token_ge:
                case token_gt:
                case token_le:
                case token_lt:
                    current = current->parent;
                    continue;
                case token_lbrace:
                    break;
                default:
                    ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                                "Invalid expression \"%s\" in
file %s",
                                expr, r->filename);
                    ap_rputs(error, r);
                    goto RETURN;
                }
                break;
            }
            if (current == (struct parse_node *) NULL) {
                new->left = root;
                new->left->parent = new;
                new->parent = (struct parse_node *) NULL;
                root = new;
            }
            else {
                new->left = current->right;
                current->right = new;
                new->parent = current;
            }
            current = new;
            break;

        case token_not:
#ifdef DEBUG_INCLUDE
            ap_rputs(" Token:  not\n", r);
#endif
            if (current == (struct parse_node *) NULL) {
                root = current = new;
                break;
            }
            /* Percolate upwards */
```

```
            while (current != (struct parse_node *) NULL) {
                switch (current->token.type) {
                case token_not:
                case token_eq:
                case token_ne:
                case token_and:
                case token_or:
                case token_lbrace:
                case token_ge:
                case token_gt:
                case token_le:
                case token_lt:
                    break;
                default:
                    ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                                "Invalid expression \"%s\" in
file %s",
                                expr, r->filename);
                    ap_rputs(error, r);
                    goto RETURN;
                }
                break;
            }
            if (current == (struct parse_node *) NULL) {
                new->left = root;
                new->left->parent = new;
                new->parent = (struct parse_node *) NULL;
                root = new;
            }
            else {
                new->left = current->right;
                current->right = new;
                new->parent = current;
            }
            current = new;
            break;

        case token_eq:
        case token_ne:
        case token_ge:
        case token_gt:
        case token_le:
        case token_lt:
```

```
#ifdef DEBUG_INCLUDE
            ap_rputs(" Token:  eq/ne/ge/gt/le/lt\n", r);
#endif
            if (current == (struct parse_node *) NULL) {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "Invalid expression \"%s\" in file
%s",
                            expr, r→filename);
                ap_rputs(error, r);
                goto RETURN;
            }
            /* Percolate upwards */
            while (current != (struct parse_node *) NULL) {
                switch (current→token.type) {
                case token_string:
                case token_group:
                    current = current→parent;
                    continue;
                case token_lbrace:
                case token_and:
                case token_or:
                    break;
                case token_not:
                case token_eq:
                case token_ne:
                case token_ge:
                case token_gt:
                case token_le:
                case token_lt:
                default:
                    ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                                "Invalid expression \"%s\" in
file %s",
                                expr, r→filename);
                    ap_rputs(error, r);
                    goto RETURN;
                }
                break;
            }
            if (current == (struct parse_node *) NULL) {
                new→left = root;
                new→left→parent = new;
```

```
                new->parent = (struct parse_node *) NULL;
                root = new;
            }
            else {
                new->left = current->right;
                current->right = new;
                new->parent = current;
            }
            current = new;
            break;

        case token_rbrace:
#ifdef DEBUG_INCLUDE
            ap_rputs(" Token:  rbrace\n", r);
#endif
            while (current != (struct parse_node *) NULL) {
                if (current->token.type == token_lbrace) {
                    current->token.type = token_group;
                    break;
                }
                current = current->parent;
            }
            if (current == (struct parse_node *) NULL) {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "Unmatched ')' in \"%s\" in file
%s",
                            expr, r->filename);
                ap_rputs(error, r);
                goto RETURN;
            }
            break;

        case token_lbrace:
#ifdef DEBUG_INCLUDE
            ap_rputs(" Token:  lbrace\n", r);
#endif
            if (current == (struct parse_node *) NULL) {
                root = current = new;
                break;
            }
            /* Percolate upwards */
            while (current != (struct parse_node *) NULL) {
                switch (current->token.type) {
```

```
                case token_not:
                case token_eq:
                case token_ne:
                case token_and:
                case token_or:
                case token_lbrace:
                case token_ge:
                case token_gt:
                case token_le:
                case token_lt:
                    break;
                case token_string:
                case token_group:
                default:
                    ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                                "Invalid expression \"%s\" in
file %s",
                                expr, r->filename);
                    ap_rputs(error, r);
                    goto RETURN;
                }
                break;
            }
            if (current == (struct parse_node *) NULL) {
                new->left = root;
                new->left->parent = new;
                new->parent = (struct parse_node *) NULL;
                root = new;
            }
            else {
                new->left = current->right;
                current->right = new;
                new->parent = current;
            }
            current = new;
            break;
        default:
            break;
        }
    }

    /* Evaluate Parse Tree */
    current = root;
```

```
    while (current != (struct parse_node *) NULL) {
        switch (current->token.type) {
        case token_string:
#ifdef DEBUG_INCLUDE
            ap_rputs("    Evaluate string\n", r);
#endif
            parse_string(r, current->token.value, buffer, sizeof(buffer), 0,
dep_table);
            ap_cpystrn(current->token.value, buffer,
sizeof(current->token.value));
            current->value = (current->token.value[0] != '\0');
            current->done = 1;
            current = current->parent;
            break;

        case token_and:
        case token_or:
#ifdef DEBUG_INCLUDE
            ap_rputs("    Evaluate and/or\n", r);
#endif
            if (current->left == (struct parse_node *) NULL ||
                current->right == (struct parse_node *) NULL) {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "Invalid expression \"%s\" in file
%s",
                            expr, r->filename);
                ap_rputs(error, r);
                goto RETURN;
            }
            if (!current->left->done) {
                switch (current->left->token.type) {
                case token_string:
                    parse_string(r, current->left->token.value,
                                 buffer, sizeof(buffer), 0, dep_table);
                    ap_cpystrn(current->left->token.value, buffer,
                            sizeof(current->left->token.value));
                    current->left->value =
(current->left->token.value[0] != '\0');
                    current->left->done = 1;
                    break;
                default:
                    current = current->left;
                    continue;
```

```
                }
            }
            if (!current->right->done) {
                switch (current->right->token.type) {
                case token_string:
                    parse_string(r, current->right->token.value,
                                 buffer, sizeof(buffer), 0, dep_table);
                    ap_cpystrn(current->right->token.value, buffer,
                            sizeof(current->right->token.value));
                    current->right->value =
(current->right->token.value[0] != '\0');
                    current->right->done = 1;
                    break;
                default:
                    current = current->right;
                    continue;
                }
            }
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, "     Left: ", current->left->value ? "1" : "0",
                   "\n", NULL);
            ap_rvputs(r, "     Right: ", current->right->value ? "1" :
"0",
                   "\n", NULL);
#endif
            if (current->token.type == token_and) {
                current->value = current->left->value &&
current->right->value;
            }
            else {
                current->value = current->left->value ||
current->right->value;
            }
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, "     Returning ", current->value ? "1" : "0",
                   "\n", NULL);
#endif
            current->done = 1;
            current = current->parent;
            break;

        case token_eq:
        case token_ne:
#ifdef DEBUG_INCLUDE
```

```
            ap_rputs(" Evaluate eq/ne\n", r);
#endif
            if ((current->left == (struct parse_node *) NULL) ||
                (current->right == (struct parse_node *) NULL) ||
                (current->left->token.type != token_string) ||
                (current->right->token.type != token_string)) {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "Invalid expression \"%s\" in file
%s",
                            expr, r->filename);
                ap_rputs(error, r);
                goto RETURN;
            }
            parse_string(r, current->left->token.value,
                         buffer, sizeof(buffer), 0, dep_table);
            ap_cpystrn(current->left->token.value, buffer,
                        sizeof(current->left->token.value));
            parse_string(r, current->right->token.value,
                         buffer, sizeof(buffer), 0, dep_table);
            ap_cpystrn(current->right->token.value, buffer,
                        sizeof(current->right->token.value));
            if (current->right->token.value[0] == '/') {
                int len;
                len = strlen(current->right->token.value);
                if (current->right->token.value[len - 1] == '/') {
                    current->right->token.value[len - 1] = '\0';
                }
                else {
                    ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                                "Invalid rexp \"%s\" in file
%s",
                                current->right->token.value,
r->filename);
                    ap_rputs(error, r);
                    goto RETURN;
                }
#ifdef DEBUG_INCLUDE
                ap_rvputs(r, " Re Compare (",
current->left->token.value,
                  ") with /", &current->right->token.value[1], "/\n",
NULL);
#endif
```

```
                    current→value =
                        re_check(r, current→left→token.value,
                                 &current→right→token.value[1]);
                }
                else {
#ifdef DEBUG_INCLUDE
                    ap_rvputs(r, " Compare (", current→left→token.value,
                            ") with (", current→right→token.value,
")\n", NULL);
#endif
                    current→value =
                        (strcmp(current→left→token.value,
                                current→right→token.value) == 0);
                }
                if (current→token.type == token_ne) {
                    current→value = !current→value;
                }
#ifdef DEBUG_INCLUDE
                ap_rvputs(r, " Returning ", current→value ? "1" : "0",
                        "\n", NULL);
#endif
                current→done = 1;
                current = current→parent;
                break;
            case token_ge:
            case token_gt:
            case token_le:
            case token_lt:
#ifdef DEBUG_INCLUDE
                ap_rputs(" Evaluate ge/gt/le/lt\n", r);
#endif
                if ((current→left == (struct parse_node *) NULL) ||
                    (current→right == (struct parse_node *) NULL) ||
                    (current→left→token.type != token_string) ||
                    (current→right→token.type != token_string)) {
                    ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                                "Invalid expression \"%s\" in file
%s",
                                expr, r→filename);
                    ap_rputs(error, r);
                    goto RETURN;
                }
                parse_string(r, current→left→token.value,
```

```
                        buffer, sizeof(buffer), 0, dep_table);
            ap_cpystrn(current->left->token.value, buffer,
                       sizeof(current->left->token.value));
            parse_string(r, current->right->token.value,
                        buffer, sizeof(buffer), 0, dep_table);
            ap_cpystrn(current->right->token.value, buffer,
                       sizeof(current->right->token.value));
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, " Compare (", current->left->token.value,
                   ") with (", current->right->token.value, ")\n",
NULL);
#endif
            current->value =
                strcmp(current->left->token.value,
                       current->right->token.value);
            if (current->token.type == token_ge) {
                current->value = current->value >= 0;
            }
            else if (current->token.type == token_gt) {
                current->value = current->value > 0;
            }
            else if (current->token.type == token_le) {
                current->value = current->value <= 0;
            }
            else if (current->token.type == token_lt) {
                current->value = current->value < 0;
            }
            else {
                current->value = 0; /* Don't return -1 if unknown token
*/
            }
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, " Returning ", current->value ? "1" : "0",
                   "\n", NULL);
#endif
            current->done = 1;
            current = current->parent;
            break;

        case token_not:
            if (current->right != (struct parse_node *) NULL) {
                if (!current->right->done) {
                    current = current->right;
                    continue;
```

```
                }
                current->value = !current->right->value;
            }
            else {
                current->value = 0;
            }
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, " Evaluate !:  ", current->value ? "1" : "0",
                    "\n", NULL);
#endif
            current->done = 1;
            current = current->parent;
            break;

        case token_group:
            if (current->right != (struct parse_node *) NULL) {
                if (!current->right->done) {
                    current = current->right;
                    continue;
                }
                current->value = current->right->value;
            }
            else {
                current->value = 1;
            }
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, " Evaluate ():  ", current->value ? "1" :
"0",
                    "\n", NULL);
#endif
            current->done = 1;
            current = current->parent;
            break;

        case token_lbrace:
            ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                        "Unmatched '(' in \"%s\" in file %s",
                        expr, r->filename);
            ap_rputs(error, r);
            goto RETURN;

        case token_rbrace:
            ap_log_rerror(APLOG_MARK,
```

```
APLOG_NOERRNO|APLOG_ERR, r,
                        "Unmatched ')' in \"%s\" in file %s",
                        expr, r->filename);
            ap_rputs(error, r);
            goto RETURN;

        default:
            ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                        "bad token type");
            ap_rputs(error, r);
            goto RETURN;
        }
    }

    retval = (root == (struct parse_node *) NULL) ? 0 : root->value;
  RETURN:
    ap_destroy_pool(expr_pool);
    return (retval);
}

static int handle_if(FILE *in, request_rec *r, const char *error,
                     int *conditional_status, int *printing,
                     dep_table_t *dep_table)
{
    char tag[MAX_STRING_LEN];
    char *tag_val;
    char *expr;

    expr = NULL;
    while (1) {
        tag_val = get_tag(r->pool, in, tag, sizeof(tag), 0);
        if (*tag == '\0') {
            return 1;
        }
        else if (!strcmp(tag, "done")) {
            if (expr == NULL) {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "missing expr in if statement:  %s",
                            r->filename);
                ap_rputs(error, r);
                return 1;
            }
```

```
            *printing = *conditional_status = parse_expr(r, expr, error,
                                                        dep_table);
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, "**** if conditional_status=\"",
                *conditional_status ? "1" : "0", "\"\n", NULL);
#endif
            return 0;
        }
        else if (!strcmp(tag, "expr")) {
            expr = tag_val;
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, "**** if expr=\"", expr, "\"\n", NULL);
#endif
        }
        else {
            ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                    "unknown parameter \"%s\" to tag if in
%s",
                    tag, r->filename);
            ap_rputs(error, r);
        }
    }
}

static int handle_elif(FILE *in, request_rec *r, const char *error,
                       int *conditional_status, int *printing,
                       dep_table_t *dep_table )
{
    char tag[MAX_STRING_LEN];
    char *tag_val;
    char *expr;

    expr = NULL;
    while (1) {
        tag_val = get_tag(r->pool, in, tag, sizeof(tag), 0);
        if (*tag == '\0') {
            return 1;
        }
        else if (!strcmp(tag, "done")) {
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, "**** elif conditional_status=\"",
                *conditional_status ? "1" : "0", "\"\n", NULL);
#endif
```

```
            if (*conditional_status) {
                *printing = 0;
                return (0);
            }
            if (expr == NULL) {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "missing expr in elif statement:
%s",
                            r->filename);
                ap_rputs(error, r);
                return 1;
            }
            *printing = *conditional_status = parse_expr(r, expr, error,
                                                         dep_table);
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, "**** elif conditional_status=\"",
                   *conditional_status ? "1" : "0", "\"\n", NULL);
#endif
            return 0;
        }
        else if (!strcmp(tag, "expr")) {
            expr = tag_val;
#ifdef DEBUG_INCLUDE
            ap_rvputs(r, "**** if expr=\"", expr, "\"\n", NULL);
#endif
        }
        else {
            ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                        "unknown parameter \"%s\" to tag if in
%s",
                        tag, r->filename);
            ap_rputs(error, r);
        }
    }
}

static int handle_else(FILE *in, request_rec *r, const char *error,
                       int *conditional_status, int *printing)
{
    char tag[MAX_STRING_LEN];

    if (!get_tag(r->pool, in, tag, sizeof(tag), 1)) {
```

```
        return 1;
    }
    else if (!strcmp(tag, "done")) {
#ifdef DEBUG_INCLUDE
        ap_rvputs(r, "**** else conditional_status=\"",
                *conditional_status ? "1" : "0", "\"\n", NULL);
#endif
        *printing = !(*conditional_status);
        *conditional_status = 1;
        return 0;
    }
    else {
        ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                    "else directive does not take tags in %s",
                    r->filename);
        if (*printing) {
            ap_rputs(error, r);
        }
        return -1;
    }
}

static int handle_endif(FILE *in, request_rec *r, const char *error,
                        int *conditional_status, int *printing)
{
    char tag[MAX_STRING_LEN];

    if (!get_tag(r->pool, in, tag, sizeof(tag), 1)) {
        return 1;
    }
    else if (!strcmp(tag, "done")) {
#ifdef DEBUG_INCLUDE
        ap_rvputs(r, "**** endif conditional_status=\"",
                *conditional_status ? "1" : "0", "\"\n", NULL);
#endif
        *printing = 1;
        *conditional_status = 1;
        return 0;
    }
    else {
        ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                    "endif directive does not take tags in %s",
```

```
                    r→filename);
            ap_rputs(error, r);
            return -1;
        }
}

static int handle_set(FILE *in, request_rec *r, const char *error,
                      dep_table_t *dep_table, FILE *cf)
{
    char tag[MAX_STRING_LEN];
    char parsed_string[MAX_STRING_LEN];
    char *tag_val;
    char *var;

    var = (char *) NULL;
    while (1) {
        if (!(tag_val = get_tag(r→pool, in, tag, sizeof(tag), 1))) {
            return 1;
        }
        else if (!strcmp(tag, "done")) {
            return 0;
        }
        else if (!strcmp(tag, "var")) {
            var = tag_val;
        }
        else if (!strcmp(tag, "value")) {
            if (var == (char *) NULL) {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "variable must precede value in set
directive in %s",
                            r→filename);
                ap_rputs(error, r);
                fputs(error, cf);
                return -1;
            }
            parse_string(r, tag_val, parsed_string, sizeof(parsed_string),
0, dep_table);
            ap_table_setn(r→subprocess_env, var, ap_pstrdup(r→pool,
parsed_string));
            ap_table_setn(r→notes, var, get_id(r) );
        }
        else {
            ap_log_rerror(APLOG_MARK,
```

```
APLOG_NOERRNO|APLOG_ERR, r,
                        "Invalid tag for set directive in %s",
r->filename);
            ap_rputs(error, r);
            fputs(error,cf);
            return -1;
        }
    }
}

static int handle_printenv(FILE *in, request_rec *r, const char *error,
                       dep_table_t *dep_table, FILE *cf)
{
    char tag[MAX_STRING_LEN];
    char *tag_val;
    array_header *arr = ap_table_elts(r->subprocess_env);
    table_entry *elts = (table_entry *) arr->elts;
    int i;

    set_no_cache( dep_table );
    if (!(tag_val = get_tag(r->pool, in, tag, sizeof(tag), 1))) {
        return 1;
    }
    else if (!strcmp(tag, "done")) {
        for (i = 0; i < arr->nelts; ++i) {
            char *key = ap_escape_html(r->pool, elts[i].key);
            char *value = ap_escape_html( r->pool, elts[i].val);
            ap_rvputs(r, key, "=", value, "\n", NULL);
            fprintf(cf, "%s=%s\n", key, value );
        }
        return 0;
    }
    else {
        ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                    "printenv directive does not take tags in
%s",
                    r->filename);
        ap_rputs(error, r);
        fputs(error,cf);
        return -1;
    }
}
```

```
/* ------------ The main function ------------- */

/* This is a stub which parses a file descriptor. */

static void send_parsed_content(FILE *f, request_rec *r,
                                dep_table_t *dep_table, FILE *cf)
{
    char directive[MAX_STRING_LEN], error[MAX_STRING_LEN];
    char timefmt[MAX_STRING_LEN];
    int noexec = ap_allow_options(r) & OPT_INCNOEXEC;
    int ret, sizefmt;
    int if_nesting;
    int printing;
    int conditional_status;

    ap_cpystrn(error, DEFAULT_ERROR_MSG, sizeof(error));
    ap_cpystrn(timefmt, DEFAULT_TIME_FORMAT, sizeof(timefmt));
    sizefmt = SIZEFMT_KMG;

/* Turn printing on */
    printing = conditional_status = 1;
    if_nesting = 0;

#ifndef WIN32
    ap_chdir_file(r->filename);
#endif
    if (r->args) { /* add QUERY stuff to env cause it ain't yet */
        char *arg_copy = ap_pstrdup(r->pool, r->args);

        ap_table_setn(r->subprocess_env, "QUERY_STRING", r->args);
        ap_unescape_url(arg_copy);
        ap_table_setn(r->subprocess_env, "QUERY_STRING_UNESCAPED",
                  ap_escape_shell_cmd(r->pool, arg_copy));
    }

    while (1) {
        if (!find_string(f, STARTING_SEQUENCE, r, printing, cf)) {
            if (get_directive(f, directive, sizeof(directive), r->pool)) {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                            "mod_include:  error reading
directive in %s",
```

```
                r->filename);
    ap_rputs(error, r);
    fputs(error,cf);
    return;
}
if (!strcmp(directive, "if")) {
    if (!printing) {
        if_nesting++;
    }
    else {
        ret = handle_if(f, r, error, &conditional_status,
                        &printing, dep_table);
        if_nesting = 0;
    }
    continue;
}
else if (!strcmp(directive, "else")) {
    if (!if_nesting) {
      ret = handle_else(f, r, error, &conditional_status,
                        &printing);
    }
    continue;
}
else if (!strcmp(directive, "elif")) {
    if (!if_nesting) {
        ret = handle_elif(f, r, error, &conditional_status,
                          &printing, dep_table);
    }
    continue;
}
else if (!strcmp(directive, "endif")) {
    if (!if_nesting) {
      ret = handle_endif(f, r, error, &conditional_status,
                         &printing);
    }
    else {
        if_nesting--;
    }
    continue;
}
if (!printing) {
    continue;
}
if (!strcmp(directive, "exec")) {
```

```
                if (noexec) {
                    ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                                  "exec used but not allowed in
%s",
                                  r->filename);
                    if (printing) {
                        ap_rputs(error, r);
                    }
                    ret = find_string(f, ENDING_SEQUENCE, r, 0, cf);
                }
                else {
                    ret = handle_exec(f, r, error, dep_table, cf);
                }
            }
            else if (!strcmp(directive, "config")) {
                ret = handle_config(f, r, error, timefmt, &sizefmt,
                                    dep_table, cf);
            }
            else if (!strcmp(directive, "set")) {
              ret = handle_set(f, r, error, dep_table, cf);
            }
            else if (!strcmp(directive, "include")) {
              ret = handle_include(f, r, error, noexec, dep_table, cf);
            }
            else if (!strcmp(directive, "echo")) {
              ret = handle_echo(f, r, error, dep_table, cf);
            }
            else if (!strcmp(directive, "fsize")) {
                ret = handle_fsize(f, r, error, sizefmt, dep_table, cf);
            }
            else if (!strcmp(directive, "flastmod")) {
                ret = handle_flastmod(f, r, error, timefmt, dep_table, cf);
            }
            else if (!strcmp(directive, "printenv")) {
                ret = handle_printenv(f, r, error, dep_table, cf);
            }
            else {
                ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                              "unknown directive \"%s\" "
                              "in parsed doc %s",
                              directive, r->filename);
                if (printing) {
```

```
                        ap_rputs(error, r);
                        fputs(error,cf);
                    }
                    ret = find_string(f, ENDING_SEQUENCE, r, 0, cf);
                }
                if (ret) {
                    ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                                "premature EOF in parsed file %s",
                                r->filename);
                    return;
                }
            }
            else {
                return;
            }
        }
}

/*****************************************************************
 *
 * XBITHACK. Sigh... NB it's configurable per-directory; the compile-time
 * option only changes the default.
 */

module includes_module;
enum xbithack {
    xbithack_off, xbithack_on, xbithack_full
};

#ifdef XBITHACK
#define DEFAULT_XBITHACK xbithack_full
#else
#define DEFAULT_XBITHACK xbithack_off
#endif

static void *create_includes_dir_config(pool *p, char *dummy)
{
    enum xbithack *result = (enum xbithack *) ap_palloc(p,
sizeof(enum xbithack));
    *result = DEFAULT_XBITHACK;
    return result;
}
```

```
static const char *set_xbithack(cmd_parms *cmd, void *xbp, char *arg)
{
    enum xbithack *state = (enum xbithack *) xbp;

    if (!strcasecmp(arg, "off")) {
        *state = xbithack_off;
    }
    else if (!strcasecmp(arg, "on")) {
        *state = xbithack_on;
    }
    else if (!strcasecmp(arg, "full")) {
        *state = xbithack_full;
    }
    else {
        return "XBitHack must be set to Off, On, or Full";
    }

    return NULL;
}

static int send_parsed_file(request_rec *r)
{
    FILE *f;
    svr_config *cfg = get_server_config( r->server );
    enum xbithack *state = get_directory_config( r );
    int errstatus;
    request_rec *parent;
    time_t req_time;

    if (!(ap_allow_options(r) & OPT_INCLUDES)) {
        return DECLINED;
    }
    r->allowed |= (1 << M_GET);
    if (r->method_number != M_GET) {
        return DECLINED;
    }
    if (r->finfo.st_mode == 0) {
        ap_log_rerror(APLOG_MARK,
APLOG_NOERRNO|APLOG_ERR, r,
                      "File does not exist:  %s",
                      (r->path_info
                       ? ap_pstrcat(r->pool, r->filename, r->path_info,
NULL)
                       : r->filename));
```

```
        return HTTP_NOT_FOUND;
    }

    if (!(f = ap_pfopen(r→pool, r→filename, "r"))) {
        ap_log_rerror(APLOG_MARK, APLOG_ERR, r,
                    "file permissions deny server access:  %s",
r→filename);
        return HTTP_FORBIDDEN;
    }

    if ((*state == xbithack_full)
#if !defined(OS2) && !defined(WIN32)
    /* OS/2 dosen't support Groups. */
        && (r→finfo.st_mode & S_IXGRP)
#endif
        ) {
        ap_update_mtime(r, r→finfo.st_mtime);
        ap_set_last_modified(r);
    }
    if ((errstatus = ap_meets_conditions(r)) != OK) {
        return errstatus;
    }

    ap_send_http_header(r);

    if (r→header_only) {
        ap_pfclose(r→pool, f);
        return OK;
    }

#define SUB_REQ_STRING "Sub request to mod_include"
#define PARENT_STRING "Parent request to mod_include"

    if (ap_table_get(r→notes, SUB_REQ_STRING)) {
        /*
         * The note is a flag to mod_include that this request
         * should be treated as if it was a subrequest originating
         * in the handle_include() procedure of mod_include.
         */

        /*
         * There is no good way to pass the parent request_rec to
mod_include.
         * Tables only take string values and there is nowhere appropriate
```

```
in
         * in the request_rec that can safely be used. So, search for the
         * parent note by walking up the r->main list of subrequests, and
at
         * each level walking back through any internal redirects. This is
         * the same request walking that mod_include uses in the procedure
         * handle_include().
         */
        request_rec *p = r->main;
        request_rec *q = p;

        while (q) {
            if (ap_table_get(q->notes, PARENT_STRING)) {
                /* Kludge — See below */
                ap_set_module_config(r->request_config,
&includes_module, q);

                /* Create the initial environment in the parent */
                ap_add_common_vars(q);
                ap_add_cgi_vars(q);
                add_include_vars(q, DEFAULT_TIME_FORMAT);

                /* Cleanup - This should allow this technique to nest */
                ap_table_unset(r->notes, SUB_REQ_STRING);
                ap_table_unset(q->notes, PARENT_STRING);
                break;
            }
            if (q->prev != NULL) {
                q = q->prev;
            }
            else {
                p = p->main;
                q = p;
            }
        }
    }

    if ((parent = ap_get_module_config(r->request_config,
&includes_module))) {
        /* Kludge — for nested includes, we want to keep the subprocess
         * environment of the base document (for compatibility); that
means
         * torquing our own last_modified date as well so that the
         * LAST_MODIFIED variable gets reset to the proper value if the
```

```
         * nested document resets <!--#config timefmt-->.
         * We also insist that the memory for this subrequest not be
         * destroyed, that's dealt with in handle_include().
         */
        r->subprocess_env = parent->subprocess_env;
        r->notes = parent->notes;
        ap_pool_join(parent->pool, r->pool);
        r->finfo.st_mtime = parent->finfo.st_mtime;
    }
    else {
        /* we're not a nested include, so we create an initial
         * environment */
        ap_add_common_vars(r);
        ap_add_cgi_vars(r);
        add_include_vars(r, DEFAULT_TIME_FORMAT);
    }
    /* XXX: this is bogus, at some point we're going to do a subrequest,
     * and when we do it we're going to be subjecting code that doesn't
     * expect to be signal-ready to SIGALRM. There is no clean way to
     * fix this, except to put alarm support into BUFF. -djg
     */
    ap_hard_timeout("send SSI", r);

#ifdef CHARSET_EBCDIC
    /* XXX:@@@ Is the generated/included output ALWAYS in
text/ebcdic format? */
    ap_bsetflag(r->connection->client, B_EBCDIC2ASCII, 1);
#endif

    /*******************************************************************
     * Caching implemented here *
     *******************************************************************/
    if( (req_time = is_cache_valid( r, cfg )) != -1 ){
      ap_rputs( "<b>From cache</b><br>\n", r );
      send_cached_page( r, cfg );
    }
    else{
      char *dep_file, *cache_file;
      FILE *cf = NULL;
      dep_table_t *dep_table;
      ap_send_http_header(r);
      /* Get filenames */
      dep_file = get_dep_file( r, cfg );
      cache_file = get_cache_file( r, cfg );
```

```c
        /* Init dep_table */
        dep_table = make_dep_table( r );
        add_dep_entry( dep_table, r->filename );
        /* Redirect output to file */
        unlink( cache_file );
        cf = ap_pfopen( r->pool, cache_file, "w" );
        /* Create content */
        ap_rputs( "<b>From source</b><br>\n", r );
        send_parsed_content(f, r, dep_table, cf);
        /* Restore output pipe */
        ap_pfclose( r->pool, cf );
        /* Generate dep file if possible */
        if( dep_table->can_cache == 1){
          write_dep_table( dep_table, dep_file );
        }
    }

    if (parent) {
        /*
         * All the work is finished for this subrequest. The following
         * makes it safe for the creator of the subrequest to destroy it
         * via ap_destroy_sub_req() once the call to ap_run_sub_req()
         * returns. This is required since the original pool of the
         * subrequest has been merged into the pool of the parent request
         * of the subrequest (see Kludge above). The alternative is to
         * NOT destroy the subrequest.
         */
        r->pool = ap_make_sub_pool(r->pool);
    }

    ap_kill_timeout(r);
    return OK;
}

static int send_shtml_file(request_rec *r)
{
    r->content_type = "text/html";
    return send_parsed_file(r);
}

static int xbithack_handler(request_rec *r)
{
#if defined(OS2) || defined(WIN32)
    /* OS/2 dosen't currently support the xbithack. This is being worked
```

```
on. */
    return DECLINED;
#else
    enum xbithack *state;

    if (!(r->finfo.st_mode & S_IXUSR)) {
        return DECLINED;
    }

    state = (enum xbithack *) ap_get_module_config(r->per_dir_config,
                                                   &includes_module);

    if (*state == xbithack_off) {
        return DECLINED;
    }
    return send_parsed_file(r);
#endif
}

static const command_rec includes_cmds[ ] =
{
    {"XBitHack", set_xbithack, NULL, OR_OPTIONS, TAKE1, "Off,
On, or Full"},
    {NULL}
};

static const handler_rec includes_handlers[ ] =
{
    {INCLUDES_MAGIC_TYPE, send_shtml_file},
    {INCLUDES_MAGIC_TYPE3, send_shtml_file},
    {"server-parsed", send_parsed_file},
    {"text/html", xbithack_handler},
    {NULL}
};

module MODULE_VAR_EXPORT includes_module =
{
    STANDARD_MODULE_STUFF,
    NULL, /* initializer */
    create_includes_dir_config, /* dir config creater */
    NULL, /* dir merger — default is to override */
    create_includes_svr_config, /* server config */
    NULL, /* merge server config */
    includes_cmds, /* command table */
```

```
    includes_handlers, /* handlers */
    NULL, /* filename translation */
    NULL, /* check_user_id */
    NULL, /* check auth */
    NULL, /* check access */
    NULL, /* type_checker */
    NULL, /* fixups */
    NULL, /* logger */
    NULL, /* header parser */
    NULL, /* child_init */
    NULL, /* child_exit */
    NULL /* post read-request */
};
```