Gcc Compile Server

Per Bothner
Apple Computer
<pbothner@apple.com>

The way the `gcc` user-level program invokes other programs (such as `cc1`, `as`, and `ld`) to compile programs has changed little over the years. Except for the recent integration of the C pre-processor `cpp` with the compiler proper `cc1`, it works very much like the original Bell Labs K+R C compiler: The `gcc` driver runs a fresh `cc1`/`cc1plus`/... executable for each source C/C++/... program that needs to be compiled, reading a single input source file, and writing a single assembler output file.

This model has (at least) two big disadvantages:

- Compiling or re-compiling many files is slow. Most obviously there is the the overhead of repeatedly creating a fresh executable. Even more significant is that each included header file has to be re-read from scratch for each main file. This is a big problem especially for C++, and has lead to work-arounds like pre-compiled header files.
- When compiling a source file the compiler has no knowledge of what is in other source files. This limits the opportunities for "cross-module" (or "whole-program") optimization, such as inter-module inlining.

The compile server project improves on these problems as follows:

- The can be more than one input files to a compilation, and they are compiled together to a single output file. It can create tree representation for all the input files, and delay code generation and optimizations such as inlining until it has read all the input files.
- The compiler can be invoked in *server mode*, in which case it enters a loop, waiting for compilation requests. Each request specifies the name of one or more input files to compile, and the name of a requested output assembler file. When the compiler is done with one file, it does some cleaning up, and then waits for the next compilation request.

We will primarily discuss the latter server mode, but multiple-file-compilation is relevant to this discussion because both mechanisms require changing the logic and control flow in the compiler proper.

The compile server compiles multiple source files, without any extra `fork`ing or `exec`ing. This provides some speedup, and so does having to only once initialize tables and built-in declarations. However, the substantial speed-up comes from processing each header file only once. The current work concentrates on front-ends that make use of `cpplib`, i.e. the C family of languages. The goal is to achieve performance comparable or better than with pre-compiled headers, but without having to create or manage PCHs. You are also a lot more flexible in terms of order of reading header files. Specifically, the goal is to avoid re-parsing the same header files many time, by re-using the tree nodes over multiple compilations. Similar ideas can benefit other languages (such as Java) that import declarations from external modules (or classes).

This paper describes highly experimental work-in-progress. The current prototype handles C tolerably well, and handles some non-trivial C++ packages.

The compile server (as currently implemented) uses the same working directory and command line flags (such as `-I` and `-D`) for all compilation requests.

1 Invoking the compiler

The `gcc` user-level driver takes a command line with some number of flags, one or more input file names,

and optionally an output file name. It uses a fairly complex set of rules to select which other applications it needs to run. One of these is the compiler "proper", which for C is the `cc1` program. The driver executes `cc1` once for each C input file name, creating an assembler file each time. The driver may then invoke the assembler once for each assembly file, creating relocatable binary files, which may then be linked together forming an executable or a shared library.

One part of this project is to change the `gcc` driver so that when it is asked to compile multiple C source files it just call `cc1` once, passing all the input files names to `cc1`. The latter also had to be changed so it could handle multiple input file names, compile them all, and create a single output file. This potentially speeds up compilation time, but more importantly it enables cross-module optimizations such as inter-module inlining.

Handling multiple input files is valuable, but doesn't help much with interactive development, where there are typically many frequent debug-edit-compile cycles. It would speed things up if the compiler could remember state from previous compiles between compilations. Another issue concerns existing `Makefile` scripts, which often use a separate `gcc` command for each source files. Therefore we need an actual *server*, which sits around in the background waiting for compilation requests. We want to use the existing `gcc` command-line interface so we don't have to re-write existing `Makefile`s, except that an environment variable or a single flag will request that `gcc` use a compile server. Then you can just do:

```
make CC='gcc --server'
```

2 Server protocol

The server uses Unix domain sockets to communicate with its clients. Using TCP/IP would be more general, and would be needed for a project where compilations are distributed to different machines. However, there are a number of existing projects and products that do distributed builds, and that is not the focus or goal of this project, so far. (Distributed compilation based on the compile server code may be an interesting future project.)

Unix domain sockets are more efficient that TCP/IP sockets, and are a good match for a non-distributed compile server. Domain sockets are bound to file names in the local file system. Each compilation uses the current working directory to resolve file names, so it makes sense for the server to bind itself to a socket in the current directory. (A future version might be able to change working directories for different compilations.) For the `cc1` compiler, the server listens on a socket bound to `./.cc1-server`.

The server is started by adding `-fserver` to the `cc1`/`cc1plus`/... command line. All options are otherwise as normal, except you leave out the names of the input and output files. The server does a `listen`, and enters a loop using `accept` to wait for connections. For each connection, it enters another loop, waiting for *server commands*. Each command is a single line, starting with a *command letter*. Following the command is a sequence of zero or more quoted string arguments. The quote character can be any byte: using a single quote `"'"` is human readable, but the `gcc` driver uses nul bytes `'\000'` since they cannot appear in arguments. Following the arguments is a newline character that terminates the command.

The following server commands are or will be supported:

F ("flags") Set or reset the command-line flags. (This is not implemented at the time of writing.) It is followed by zero or more nul-terminated flag values, terminated by a newline. Do not use this to set input or output file names. For example:

```
F\0-I/usr/include\0\0-DDEBUG=1\0\0-O2\0\n
```

T ("timeout") Followed by an integer in milliseconds. Sets the time-out duration. If no requests come in during that time, the server exits. If the timeout is 0, the server exits immediately.

I ("invalidate") Followed by a list of nul-terminated filenames. Any cached data for the named files are invalidated. Can be used by an IDE when an include file has been edited. (The server can also `stat` the files, but it may be more efficient to avoid that.)

S ("source") Followed by a file name argument, which is the name of an input source file to compile. There can by multiple `S` commands in

a row, in which case all of the input files are compiled, producing a single output file.

O ("output") Followed by a file name argument, which is the name of the output assembler file. The file names from previous `S` commands (since the last `O` command) are all compiled to produce the named output assembler file.

The server currently writes out diagnostics to its standard error, but it should instead send them back to the client using the socket, so the client can write out diagnostics on its standard error.

The client is either the `gcc` command, or some IDE. It could also be an enhanced `make`. It calls `socket`, and then attempts to `connect` to `"./.gcc-server"`. If there is no server running, it starts up a server, and tries again. (This part has not yet been implemented.)

If the `gcc` command is asked to compile multiple source files, it only opens a connection to the server once, and only sends a single `F` command. If a `-o` option is specified (and `-E` is not specified) then (as an optimization) we can have `gcc` do a multi-file compile, specifying a single `O` output file but multiple `S` input files.

3 Initialization

Initializing the compiler is relatively straightforward when compiling a single file. But a server needs three levels of initialization:

1. Initialization that only needs to be done once. For example creating the builtin type nodes, and declaring `__builtin` functions.
2. Initialization that needs to be done for each compilation request (i.e. for each output file). For example opening the assembler output file, and initializing various data structures used by the compiler back-end.
3. Initialization that needs to be done for each input file. For example making available any macros defined with `-D` command-line flags - even if a previous source file `#undef`'d it. Also clearing out any top-level declarations left over from previous source files.

The historical code base has a number of assumptions and dependencies that are no longer appropriate with the compile server. We interface between the language-independent `toplev.c` and the language front-ends uses callback functions that needed some changing: The call-backs and functions that do one-time-only initialization use the word `initially`, while the `init` is used for initialization that is done once per compilation request. For example the modified file `c-common.c` contains both `c_common_initially` and `c_common_init`.

In general, we want to do as much as possible in `initially` functions rather than `init` functions. The obvious reason is to avoid re-doing work needlessly, but there is a more important reason: The goal of the compile server is to save and re-use trees across compilations. These will make use of various builtin trees, such as `integer_type_node`. If these builtins get re-defined, then any trees that make use of them will become invalid.

The CPP functions make use of a `cpp_reader` structure that maintains the state of the pre-processor. The global `parse_in` is initialized to a `cpp_reader` instance allocated at `initially`-time. This is important, because the `cpp_reader` maintains a lot of state, including a cache of header file contents, that we want to preserve across compilations. In fact, a simple compile server that only preserves the contents of header files is one option for a less ambitious compile server.

4 Caching text vs tokens vs trees

The fundamental design question for a compile server is what state to save between compilations. Three options come to mind:

- Preserving header file text is easy to implement, especially as `cpplib` already has a cache that does this. We just need to tweak things a little bit. This is especially useful if the OS is slow in handling file lookup, doesn't handle memory-mapped files, or doesn't do a good job buffering files. Otherwise, the benefit should be minor. However, it is a modest change which should be easy to implement.
- We can also preserve the raw token streams in the header files, before macro-expansion. This

allows macros that expand differently for different compilations. However, we'd have to use some new data structure for preserving the tokens, and then feed them back to `cpplib`. Any performance advantage over preserving text is likely to be modest, and unlikely to justify the rather radical changes to `cpplib` that would seem to be required.

- Preserving the post-macro-expansion token stream seems more promising. Saving and later replaying the token stream coming out of `cpplib` doesn't appear to be very difficult, and would save the time used for re-reading and re-lexing header files, though it would not save the time spent on parsing and semantic analysis. On the other hand consistency checks and dealing with some of the ugly parts of the languages and the compiler are simpler.

- The best performance gain comes from saving and re-using the tree nodes after parsing and name lookup. This assumes that (normally) a header file consists mainly of declarations (including macro definitions), and the "meaning" of these declarations does not change across compilations. That "meaning" may depend on declarations in other header files, but the "meaning" of those declarations is also constant. (The C++ language specification enshrines something similar in the one-definition rule.)

  Thus if we parse a declaration in a header file, the result normally is a decl node or a macro. Re-parsing the same header file will result in an equivalent decl node or macro. So instead of re-parsing and creating new nodes, we can just skip parsing and re-use the old one.

  A difficultly in re-using trees is determining when it is actually safe and correct to do so, and when we have to re-parse the header file. Another complication is that the compiler modifies and merges trees after-the-fact in various ways. We will discuss these issues below.

  The current prototype takes this approach.

5 Granularity of re-parsing

We say that a header file or portion of one is *parsed* when actual characters are lexed, parsed, and semantic actions performed. A file or portion of one is *re-parsed* when the same text is parsed a second or subsequent time, either because the same file is included multiple times without guards, or because we're processing a new main file. The goal of the compile server is to minimize re-parsing text. Instead, we want to *re-use* a file or portion of a file, which means we want to achieve the semantic effects of parsing (typically creating and adding declarations into the global scope), without actually scanning or parsing the text. We say that we *process* a file or a portion of one to mean either parsing or re-using it.

We will later discuss how we can determine when it is ok to re-use (a part of) a file, and we have to re-parse it, but first let us consider granularity of re-parsing: When we need to re-parse, how much should we re-parse? The following approaches seem possible:

1. Re-read the entire header file. This is conceptually simple, since deciding whether to re-use or re-parse is decided when we see an `#include`. This avoids any complications about managing and seeking to a position within a file. However, this is not a major benefit, given that `cpplib` already caches entire header files, and seeking within a buffer is trivial. The problem with this approach is that handling conditionals within a header file is difficult. We have to decide at the beginning of the file whether any of it is invalid, and whether any conditional compilation directives may "go the other way" compared to when we originally parsed the file. This is doable, but non-trivial. Also, this approach may be excessively conservative, in that we have to invalidate too much.

2. Re-read a header file fragment between any pre-processor directives. Each header file is cached in a buffer. (This is not new with the compile server.) When a header file is re-used, we read from the saved buffer. Pre-processor directives (including conditionals) are handled in the normal way, by reading from the buffer. However, if the fragment following a directive (or the beginning of file) is valid, we we just restore its declarations, and skip ahead to the next directive (or end of file). This approach has the big advantage that we can use the existing code for evaluating and processing directives. It does have the disadvantage that we have to re-parse and re-evaluate directives, but simplicity and consistency probably is more valuable. There

is also a simplification because fragments (unlike header files) don't nest.

This is what is currently implemented.

3. Re-read a header file fragment between conditional compilation directives. It is a refinement of the previous option, except that `#define` (and `#undef`) are treated as part of a fragment, rather than delimiting fragments. A big advantage is that we can re-use macro definitions, without having to re-parse them.

   I think this may be the best approach, but I haven't explored it yet.

4. Re-read just an individual declaration. The problem with this is that we need to maintain some amount of state with each fragment, and the cost goes up if we make the fragments too small: There are usually lots of declarations. The advantage of smaller fragments is that there is less to re-parse when a declaration becomes invalid, which reduces the chance of other declarations becoming invalid. However, we expect that this will not compensate for the extra overheads, so we have not investigated this option.

Using fragments as the unit of re-parsing lets us handle cases like this easily, where we can re-use `D1`, even if we later find out we have to re-parse `D2`:

```
#if M1
D1;
#endif
#if M2
D2
#endif
```

Header guards (as shown below to protected against multiple inclusion) are no problem when using fragments. The processing of the `#include` and the header guard doesn't change when the compile server uses fragments - the only difference is how it handles the body of the header file.

```
#ifndef __H
#define __H
...
#endif
```

6 Entering and exiting fragments

The pre-processer uses callsbacks `enter_fragment` and `exit_fragment` to let the language-front-end know about the start and end of fragments. These are bounds to the functions `cb_enter_fragment` and `cb_exit_fragment` in `c-common.c`.

The preprocessor maintains a cache of header files, including their text. Each header file also gets a `struct cpp_fragment` chain. A new `cpp_fragment` is created whenever `cpplib` starts processing a fragment and there isn't already a `cpp_fragment` for that location. This is done at the start of a header file, and after each preprocessor directive. (We will probably change the code so that `#define` and `#undef` do not delimit fragments.) If the language-specific callback returns non-NULL, then the fragment has to be (re-)parsed normally. The preprocesor remember the returned pointer, and it is passed back on subsequent `enter_fragment` calls for the same `cpp_fragment`.

If `cb_enter_fragment` returns NULL, it means the fragment can be re-used. The preprocesor skips ahead to the end of the fragment, ignoring anything skipped. The `cb_enter_fragment` will have performed any semantic actions for the fragment, such restoring declarations into the top-level scope.

At the end of a fragment, `cpplib` calls the `exit_fragment` callback, which performs any language-specific actions needed if this fragment is a candidate for future re-use. Note that `exit_fragment` is not called if `enter_fragment` returned NULL.

7 Dependencies

Before we can re-use a saved fragment, we need to determine if the declarations it *depends on* have changed, When a declaration is parsed, identifiers appearing in it (such as parameter type names) are resolved using other declarations, macros, and other dependencies. So conceptually for each declaration we must remember the set of other declarations and "things" that it depends on. This is the former's *depends-on-set*. A pre-condition for re-using a declaration when compiling a new file is that any declarations it depends on also have been re-used in the

new compilation: A depended-on declaration must have been processed, or else it will not be defined, and it must not have been re-parsed, in case that defined the declarations to something new.

Consider a header file `h1.h` containing:

```
#if M1
typedef int word;
#else
typedef long word;
#endif
/* Define flags, which depends on word. */
extern word flags;
```

Assume the first time we `#include h1.h`, `M1` is true, so `word` and `flags` are defined. Assume `M1` is false the next time we `#include h1.h`, so we get the other definition of `word`. Thus the saved definition of `flags`, which depended on the old definition of `word`, needs to be invalidated, and we have to re-parse the fragment defining `flags`.

We can use a conservative approximation of the depends-on set. For example, we can for each header file remember the set of other header files it uses, where a header file uses some other header file if any declaration defined in the former header file uses any declaration in the latter header file. We can also remember dependencies at the level of header file fragments. This is the issue of the granularity of remembered dependencies (which is related to but distinct from the granularity of re-parsing). It actually has two parts: Is a depends-on-set a set of declarations, fragments, or files? How many depends-on-sets do we maintain: One for each declaration, for each fragment, or for each file?

Assuming the granularity of re-parsing is a fragment, then there is no point in maintaining a depends-on-set for each declaration. Instead we maintain a depends-on-set for each fragment, which is the union of the depends-on-sets of the declarations *provided* by the fragment.

In the current implementation the elements of a depends-on-set are fragments: I.e. a fragment has a set of other fragments that provide declarations it depends on. This is an optimization, since there is no point in separately remembering more than one declaration from the same fragment (they will all be valid or all invalid).

(However, there is a case for making the depends-on-set elements be declarations rather than fragments, because we then don't have to map from a declaration to the fragment that provided it. The current implementation adds a field to each declaration that points to the fragment that declared it, and this is wasteful. (We can also get the fragments by mapping back from the declarations line number, but this is slower, even if we change to using the line-map structures.) However, we still need an efficient way to determine if a declaration has been re-used. We can do that by looking at the declaration's name, and verifying that the name's global binding is the declaration.)

7.1 Implementation details

For efficiency, a depends-on-set is represented as a vector (currently a `TREE_VEC`, but it could be a raw C array). This is more compact than using a list, but has the complication that we don't know how big an array to allocate. To avoid excess re-allocation, we use a global array `current_fragment_deps_stack` (that we grow if needed) and a global counter `current_fragment_deps_end`, This is used for the depends-on-set of the current fragment. When we get to the end of the fragment in `cb_exit_fragment`, we allocate the fragment's depends-on-set (in the field `uses_fragments`), whose size we now know, filling it from `current_fragment_deps_stack`, and then re-setting `current_fragment_deps_end` to 0.

We need to avoid adding the same fragment multiple times to the current depend-on-set. We do that by setting a bit in the fragment when we save it in `current_fragment_deps_stack`. If the bit is set, we don't need to add it. The bit is cleared when in `cb_exit_fragment` we copy the stack into the fragment's `uses_fragments` field.

A global counter `c_timestamp` is incremented on various occasions, and used as a "clock" for various timestamps. Each fragment has two timestamps: `read_timestamp` is set when the fragment is (re-)parsed, while `include_timestamp` is set whenever the fragment is processed (parsed or re-used). Both are set on `cb_enter_fragment`. We also have a global `main_timestamp` set whenever we starting compiling a new main file. For a fragment `f` to be valid (a candidate for re-use), we require that `f.include_timestamp < main_timestamp`, otherwise the fragment has already been processed in this compilation, and re-processing it is proba-

bly an error we want to catch. We also require that for each fragment `u` in `uses_fragments` (the depends-on-set) that all of the following are true: `u->include_timestamp >= main_timestamp` (i.e. `u` has been processed in this compilation); that `u.read_timestamp != 0` (it has been parsed at some point!); and that `u.read_timestamp <= f.read_timestamp` (the most recent time `u` was parsed was before the most recent time that `f` was parsed - i.e. that `u` hasn't been re-parsed since we last used it).

7.2 Depending on the lack of a definition

One subtle complication concerns *negative dependencies*: Some code may work one way if an identifier has no binding and a different way if it has a binding.

One example (from Geoff Keating): Suppose the tag `struct x` is undefined when this is first seen:

```
// in something.h
extern int do_something (struct x *);
```

This is legal C, but the parameter type is a "local" (and useless) type, different from any global `struct x`. Next, suppose `struct x` has been declared (a forward declaration is enough) the next time this fragment is processed. In that case the parameter type of `do_something` is the global `struct x`, and so the meaning of `do_something` has changed. However, the dependency checking discussed about will not catch this, since the first time `something.h` was included there was nothing for it depend on. This particular problem will cause a warning to be written out the first time, and we can at the same time invalidate the current fragment (disabling future reuse).

However, there may be more complex problems involving negative dependencies, for example involving C++ function overloading.

8 Macro dependencies

The meaning of a fragment may also depend on the definition of macros. Consider the following:

```
char buffer[BUFSIZ];
```

If the macro `BUFSIZ` changes, then the the type of `buffer` is different, so the containing fragment would have to be invalidated.

The implementation does not yet check for macro re-definitions.

Assuming we change the implementation so that macro definitions are part of fragments, and we still store dependencies in terms of fragments depending on other fragments, then we have the basics of what we need. All that would need to be added is that when a macro is used, we note that the current fragment depends on the fragment containing the macro definition.

Which fragments get processed will also depend on macros, but since conditional compilation directives are always re-evaluated, this is not a problem.

8.1 Depending on lack of a macro bindings

We also have the issue of negative dependencies for macros: A fragment will use an identifier, and if later that identifier is bound to a macro, then the fragment will be invalid. Consider a header file `a.h`:

```
extern int i, j;
```

and a header file `b.h`:

```
inline int foo() { return i; }
```

Suppose `file1.c` does this:

```
#include "a.h"
#include "b.h"
```

and `file2.c` does this:

```
#include "a.h"
#define int size_t
#define i j
#include "b.h"
```

In file1.c the fragment b.h depends on a.h, since it used i. But the meaning of fragment b.h in file2.c is very different.

The obvious solution is for every fragment to maintain a set of identifiers that the fragments depends on not being bound to macros, and to check this list on fragment re-use. However, this is quite expensive, as fragments will often use many non-macro identifiers. Below, is a less expensive (unimplemented) solution.

8.2 Checking lack of macro bindings

Here is one solution, that is inexpensive in the common case. For each identifier we add two bits:

```
unsigned used_as_nonmacro : 1;
unsigned also_used_as_macro : 1;
```

When an identifier is referenced, and there is no macro definition for it (i.e. `#define strcmp strcmp` doesn't count), then we set the used_as_nonmacro bit. This is permanent - we never reset it.

If an identifier with the used_as_nonmacro bit gets #defined as a macro, then we also set the also_used_as_macro bit (which is also permanent). We also invalidate all fragments. We can do this by setting this global (or field in cpp_reader):

```
int first_valid_fragment_timestamp;
```

to c_timestamp. This forces all fragments to be re-read the next time they are needed.

If an identifier is referenced, and it has the also_used_as_macro bit set, then we add it to a list belonging to the current fragment. Then the next time the fragment is needed, to check validity we check the macro state of identifier on that list.

This implementation has the advantage that the common case is cheap, not requiring any extra state except two bits per identifier. (We also need space for a list header in each fragment, but it may be possible to share with some other list.). However, the rare cases get handled without excessive cost.

9 Saving and restoring bindings

While a fragment is being parsed, each language front-end is responsible for remembering the bindings (declarations etc) that are being created, so they can be restored if the fragment is re-used. The code for this is relatively independent of the rest of the compile server code, so it can be written without understanding the details of the server.

Each binding that needs to be remembered is added to the global fragment_bindings_stack, which is (currently) a TREE_VEC. How much of the stack is currently used is given by the global fragment_bindings_end. There are helper functions note_fragment_binding_1, note_fragment_binding_2, and note_fragment_binding_3 to add trees to the stack. What is added is up to the front-end; we'll give examples later. At the end of the fragment, cb_exit_fragment will allocate a TREE_VEC whose length is fragment_bindings_end, assign that to the fragments bindings field, and copy that many elements from fragment_bindings_stack.

If a fragment is re-used, then cb_enter_fragment will call the language-specific function restore_from_fragment. This is responsible for going through the bindings array and restoring the bindings.

The C language front-end currently does the following:

- pushdecl calls note_fragment_binding_1, passing it the declaration that is pushdecl's argument.
- pushtag calls note_fragment_binding_1, passing it the TREE_LIST that is used to link the type into the tag scope. This is called when the tag is declared, including forward declarations.
- finish_struct and finish_enum both call note_fragment_binding_3, passing it the struct/union/enum type, the field list or enum values list, and the type size. This is called when a struct/union/enum tag type is defined.

To restore the bindings when re-using a fragment, the function restore_from_fragment in c-decl.c just loops through the bindings TREE_VEC.

- If the element is a declaration, it set the `IDENTIFIER_GLOBAL_VALUE` of the declaration's name to point to the declaration, and chains it into the `names` list of the `current_binding_level`.
- If the element is a `TREE_LIST`, we know it was created by `pushtag`. So we chain it into the `tags` list of the `current_binding_level`. We also null out the `TYPE_FIELD` and `TYPE_SIZE` fields of the tag type, so don't get complaints if there is a later `start_struct`. This restores a tag type declaration.
- If the element is a type node, then it must have been created by `finish_struct` or `finish_enum`, and must be followed by a fields and a size node. Set the `TYPE_FIELDS` and the `TYPE_SIZE` fields of the type to those values. This restores a tag type definition.

10 Modification-in-place of trees

As the compilation proceeds, the compiler sometimes modifies existing declarations. This causes some difficulties. Some examples:

- When the C or C++ front-end sees a declaration with the same name as a previous declaration in the same scope, it calls the function `duplicate_decls` to compare the old and new declarations. This happens most frequently when the old declaration is a forward or tentative declaration. If the declarations match, `duplicate_decls` may merge the information from the new declaration into the old declaration, and then discard the new declaration. If the old declaration was in a header file that the compile server re-uses, then it will incorrectly also contain the information from the new declaration.
- In C++ functions may be overloaded. When a new function declaration overloads an older function declaration, the latter is converted to a special overload declaration. When a header file containing that declaration is re-used, we may inadvertently also get overloaded functions that aren't supposed to be visible. This may effect overload resolution, or cause future incorrect error messages.
- A header file may contain a tentative structure declaration (such as `struct T`), and a different header file may contain a definition of the `struct` with all the fields. We need to be careful that re-using the former does not re-use the latter. Worse, some C programs may re-use the same structure tag for incompatible types. (This is poor style and rare, but we should at least detect it.)

Most of these merging operations are in practice harmless, or at least will very rarely cause problems, though they may cause some errors to not be properly detected. Sometimes the merging operations can be handled by special code, or it may be possible to "clean up" the compiler to avoid them. However, there are so many places in the compiler that modify older tree nodes that we need a general framework for dealing with them. Such a framework is an *undo buffer*.

Whenever the compiler destructively modifies a tree node that "belongs" to some "other" header fragment, then it needs to append to a global undo buffer enough information to undo the modification. Before starting to compile a new main file, the compiler runs through the undo buffer in inverse order, undoing the remembered modifications. This allows fragment re-use to push the associated declarations without contamination from other fragments.

Implementation of the undo buffer has just started, so I don't know how will it will work in practice, or how much undo information is likely to be needed.

11 Some complications

Various unusual cases cause complications.

11.1 Nested #define inside declarations

On GNU/Linux `<bits/siginfo.h>` contains:

```
enum
{
  SI_ASYNCNL = -6,
# define SI_ASYNCNL     SI_ASYNCNL
  SI_SIGIO,
```

```
# define SI_SIGIO       SI_SIGIO
...
  SI_KERNEL = 0x80
#define SI_KERNEL       SI_KERNEL
};
```

This causes a problem if `#define` is the end of a fragment, since then we get a bunch of fragments that are not self-contained. If for some reason some but not all of these fragments get invalidated and have to be re-parsed, then the parser will get very confused!

This particular case is not a problem if we implement the model that `#define` is part of a fragment, rather than delimiting one, as I think we should. Another and more general solution if to invalidate a fragment if it starts or ends not at top level: I.e. nested inside some other declaration or scope. We discuss this next.

11.2 Conditional compilation inside declarations

Many systems (including GNU/Linux and Darwin) have code like the following (in `<netinet/ip.h>`):

```
struct timestamp
  {
    u_int8_t len;
    u_int8_t ptr;
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int flags:4;
    unsigned int overflow:4;
#elif __BYTE_ORDER == __BIG_ENDIAN
    unsigned int overflow:4;
    unsigned int flags:4;
#else
# error "Please fix <bits/endian.h>"
#endif
    u_int32_t data[9];
  };
```

This particular case should not be a problem in practice, since the value of `__BYTE_ORDER` is presumably not going to change. However, it is possible that the first or last fragment might becomes invalidated for some reason, causing the non-conditional parts to get re-parsed. In that case, we need to make sure that the conditional parts also get invalidated and re-parsed. (The converse could also be true, though I don't see how that could happen.)

A general solution uses a `currently_nested` variable. It is incremented when starting a declaration (such as an enum, class, template, or inline function), and decremented when exiting the declaration. If `currently_nested` is positive when either `cb_enter_fragment` or `cb_exit_fragment` is called, then the fragment is invalidated, disabling future re-use.

This should be safe, but not ideal, as `struct timestamp` would be needlessly invalidated. It would be better (though unimplemented) to treat all the fragments that contain a part of `struct timestamp` as a single unit. A *fragment group* is a minimal sequence of fragments in the same header file such that if `currently_nested` is true at the end of one fragments then it and the following fragment are both in the group. A fragment "follows" another if it is the next fragment processed during a single processing of its file. For simplicity, we require that there be no macro definitions or undefinitions within the fragment group. When we parse the fragment group, we remember all the conditionals. We treat the fragment group as a single fragment with a single constructed compound conditional. When we process the group the next time, we evaluate this compound conditional at the start of the group. If it matches, we use the fragments declarations like a normal re-use. If it does not match, we re-parse the fragments as multiple normal fragments.

11.3 Other non-nesting

One common example of non-nesting:

```
#ifdef __cplusplus
extern "{"
#endif
```

This causes the following to nested syntactically. However, we don't want it to cause following fragments to be invalidated!

C++ namespaces.may have similar issues.

11.4 Types defined in multiple locations

The C standard requires that both `<stdio.h>` and `<stdlib.h>` define `size_t`. The trick is to do this

without a duplicate definition if both are included. One common solution (used on Darwin and other *BSD system) is to define `size_t` in both headers, but use guards:

```
#ifndef __size_t_defined
#define __size_t_defined
typedef __SIZE_TYPE size_t;
#endif
```

Now suppose `a.c` has

```
#include <stdio.h>
#include <stdlib.h>
```

and `b.c` has:

```
#include <stdlib.h>
#include <stdio.h>
```

In this case the dependencies might prevent us from re-using the cached definition of `size_t`. Worse, definitions that depend on `size_t` also have to be invalidated. Note that this is not a problem of the correctness of the compile server, only its performance.

C++ has a "one-definition rule" that requires that each type declaration etc only a single definition: If different compilation units see different definitions, they must be token-by-token the same. In practice this usually means they are in the same header file, but as in the `size_t` example, that is not strictly required. However, if there are multiple definitions, they will have inconsistent source lines. If you ask an IDE for `size_t`'s definition, it will not be able to give a unique answer. This suggests that a good rule of design is the "extended one-definition rule": There should only be a single definition, at a single location in a unique header file.

The `size_t` definitions violate this extended rule. Therefore, I think the "correct" solution is to fix the headers to not do this. (We can use `fixincludes` to avoid having to change the installed headers, of course.) A simple solution is to create a header `bits/size_t.h`:

```
#ifndef _SIZE_T_H
#define _SIZE_T_H
typedef __SIZE_TYPE size_t;
#endif
```

and then have both `stdio.h` and `stdlib.h` do `#include <bits/size_t.h>`.

There are other solutions possible, but this seems the cleanest and simplest. On GNU/Linux systems using glibc, we have:

```
# define __need_size_t
# define __need_NULL
# include <stddef.h>
```

The magic `__need_size_t_` asks `stddef.h` to define `size_t` and nothing else. This satisfies the "extended one-definition rule", and I don't know any reason why it should cause problems for the compile server. It is a rather complex mechanism, though.

12 Results and conclusions

The compile server has been used to compile sets of related C files (some Apple Carbon files) and C++ (parts of the Octave mathematical library). The preliminary results have been impressive, with speeds-ups of 3x or more. However, there are a number of constructs that are not handled correctly, some planned features (such as the undo buffer) have not been implemented yet, and for some constructs it is not clear what the right solution is. So any detailed performance numbers would be premature and misleading.

Work continues on the compile server, since we at Apple believe it has great long-term potential. The latest patches are available by emailing `<per@bothner.com>`.

Thanks to the members and management of the Apple compiler group (including Ted Goldstein, Ron Price, Mike Stump, and Geoff Keating) for discussions and support of this project, .