Reprinted from the

Proceedings of the GCC Developers' Summit

June 17th–19th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Google*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Incremental Compilation for GCC

Tom Tromey
Red Hat, Inc.
tromey@redhat.com

Abstract

The incremental compiler is intended to address perennial complaints about GCC's compile time, ideally by making compile time roughly proportional to the "size" of a change. The incremental GCC runs as a server and efficiently shares declarations across compilations. Currently the C front end works incrementally; eventually the C++ front end will be converted, and the compiler will perform incremental code generation as well.

1 Overview

It is a well-known trick in C++ circles that including all the program source files into a single large source file can reduce overall build time. This observation is the foundation of the basic idea behind the incremental compiler, which is to share parsed pieces of program text across compilations. This sharing is done by running GCC as a server, much like the earlier compile-server[5] project.

Very early on, I decided that the incremental compiler must be very easy to use. I reasoned that it would increase use of the new code if users did not need to modify their project Makefiles in order to use it. A user starts the server with `gcc --server`, and then sets the environment variable `GCCSERVER`. The `gcc` driver recognizes this setting and sends jobs to the compile server for processing.

I tried to break up the project into discrete milestones, where each milestone is useful by itself. My goal here was to avoid having a single, large project on a branch for too long; each milestone ought to be merge-worthy. In order, the milestones are:

1. Server-izing, plus modifications to the C front end. This is largely complete.
2. Incremental code generation. This task is in development.
3. Modifications to the C++ front end. This item has not been started.
4. Incremental linker. This is wishful thinking.

2 Server-ization

Surprisingly few generic changes were necessary in order to turn GCC into a compile server. The core of the server is less than 1000 lines of code; this includes interrupt handling and the communication protocol. The driver changes are trivial.

A few modules did require the addition of cleanup code, though much of this is handled automatically by the garbage collector.

I also modified GCC to garbage collect `IDENTIFIER_NODE` trees. This was needed so that server memory use did not grow unduly, but it has the nice side benefit of saving a reasonable amount of memory in ordinary compilations.

The compile server tries to decouple the front end from the middle end a bit more. In particular, it delays gimplification and handoff to `cgraph` until parsing and semantic analysis are complete. It also delays any assembly output until this time; this required some minor changes in `dwarf2out.c`.

The server communication protocol is *ad hoc*. Communication is done over a unix-domain socket; a client sends command-line options and working directory, and then a request to compile. The client also sends its `stderr` file descriptor over the socket, so that messages from the compile server go to the right place.

The server sets its internal state based on the uploaded options, manages its file descriptors, and then runs the

requested compilation. Note that it is the server that starts the assembler, not the client—this is important for incremental code generation.

Currently only `cc1` can run as a server. I plan to have each compiler run as a separate server; I don't see much value in trying to fit all the front ends into a single process. The compiler base name (e.g., "cc1") is included into the connection socket's name to facilitate this.

```
struct s;                  /* hunk 1 */
struct s { int field; };   /* hunk 2 */
```

Figure 1: First Compilation

```
struct s;                  /* hunk A */
struct s { int lose; };    /* hunk B */
struct s { int field; };   /* hunk C */
```

Figure 2: Second Compilation

3 The C Front End

I chose the C front end to be the first milestone both because it is the simpler of the front ends, and because the existence of `-combine` allowed a prototype to be written easily.

The incremental compiler works by analyzing the output of the C preprocessor, looking for reuse opportunities. I investigated including the preprocessor in the model, but after much planning, this looked too complicated. Instead, I reasoned that incremental preprocessing could be implemented later and then composed with incremental parsing.

3.1 Tree Reuse

Reusing parsed declarations across different compilations is a multi-step procedure.

First, the main parsing loop makes a quick guess at the bounds of a reusable "hunk." The type of guessing, and the size of the resulting hunk, varies based on a heuristic. For text coming from files owned by the user, a hunk is a single top-level declaration, for instance a `typedef` or a function definition. For files not owned by the user,[1] a hunk's boundaries are the locations of file-change events as reported by the preprocessor. The rationale for this difference is that files not owned by the current user are less likely to change, and larger hunks yield less bookkeeping overhead.

Next, the parser computes the hunk's signature. The signature is a checksum based on the contents of the tokens in the hunk. Some relevant information about each token is included in the signature—though, notably, not the token's source location. This omission lets the compiler reuse objects after some common kinds of edits, for instance the addition of a comment.

The signature is looked up in a hash table to see whether this particular hunk has ever been parsed before. If so, we may have several candidate parsings from which to choose.

If there is a parsed form of the hunk, we must evaluate it for suitability. Any identifier looked up while parsing the body of a hunk registers a dependency in the hunk between that identifier and the object to which it resolves.

Then, when evaluating the suitability of a hunk, we perform these same lookups, which must yield the same values. This check for equality is needed to ensure both that a hunk is only reused when its dependencies are reused and that it is not reused when there is some intervening code may change the meaning of those dependencies.

For instance, consider the code in Figure 1. Here, the second hunk completes the struct declared in the first hunk. Suppose we compile this code, and then compile the code in Figure 2.

When compiling hunk A, the compiler will reuse the trees created when hunk 1 was parsed. Then when examining hunk C, the compiler will find hunk 2 in the map, since the tokens are the same. However, because hunk 2 is not reusable due to the intervention of hunk B, it is rejected by the equality check—in order to reuse the trees from hunk 2, the previous binding of `s` must be the tree from hunk 1.

Rather than checking for tree equality of prerequisites, it would be possible to check for ABI compatibility. This

[1] This is similar, but not identical, to the existing notion of system headers.

would enable more reuse, at the expense of a potentially slower check. This could also be done conditionally. For example, it may be worthwhile to check ABI compatibility for functions definitions but not other objects. It may also be worthwhile to add a special case for forward declarations of `struct` and `union` tags, as these are very common.

In essence, the prerequisites form a dependency model of the user's program. If a textual change is made which affects the tokens making up some declaration, then that declaration must be re-parsed. This will affect the values of bindings in the current compilation, thus requiring a re-parse of all declarations referring (directly or indirectly) to the changed declaration. If a modified declaration has few users, few objects will need to be re-parsed, and recompilation will be fast. If a modified declaration has many users, then more work must be done.

If a suitable parsed hunk is found, then the declarations from this hunk are mapped into the symbol table.

Otherwise, we proceed to parse the hunk. While parsing we note dependencies, and when we've finished parsing the hunk we add it to the table. We might see several hunks with the same signature but different dependencies. The table is designed to accommodate this, as this is common in typical programs.

3.2 Decl Smashing

In order to make hunk reuse work properly, I had to remove "decl smashing" from the C front end. Formerly, the C front end would "smash" together multiple declarations into the same tree object. If the incremental compiler tried to reuse such a smashed declaration, it could end up with information visible to an old compilation but not available to the current compilation.

For example, consider Figure 1 again. In the unmodified compiler, a single `RECORD_TYPE` tree is created while parsing the first hunk, and then the same tree is overwritten while parsing the second hunk.

Rather than smash declarations, the incremental front end makes a new copy for each declaration in the source. Whenever a copy is made, the compiler notes the equivalency in a table, called the "smash map." In the above example, a separate `RECORD_TYPE` tree is created in each hunk.

```
struct s;
struct s *func (void);
struct s { int field; };
int value (void) {
  return func()->field;
}
```

Figure 3: View Convert Example

While parsing expressions, we may sometimes need to refer to the smash map and insert `VIEW_CONVERT_EXPR` conversions to account for the update.[2]

For example, consider the code in Figure 3. When parsing the function `value`, the `TREE_TYPE` of `func` will refer to the first, incomplete declaration of `s`.

If left alone, this would result in a compile-time error. Instead, at this point, we look up `s` in the smash map. This yields the complete definition of `s`, and we wrap the `INDIRECT_REF` in a `VIEW_CONVERT_EXPR`, casting from the incomplete `s` to the complete variant.

This `VIEW_CONVERT_EXPR` treatment turns out to be needed in relatively few places in the front end.

Later, during gimplification, we remove the `VIEW_CONVERT_EXPR` casts again, allowing the optimizers to do their job. This is currently done by re-smashing trees in a subprocess, so as not to effect future reuse. Using a subprocess also has the nice effect of not needing to worry about resetting state in various middle- and back-end modules.

Note that reused trees may still be modified. It is difficult in GCC to ensure that any given tree is completely immutable, and so we only achieve "relative immutability." In practice this means that some fields—e.g., `TREE_USED`—can be reset and reused in each compilation, while others—roughly those corresponding to the text of the declaration—are considered untouchable. In some cases the reuse can be extreme; for instance the language-specific fields of an `IDENTIFIER_NODE` may be modified frequently during a compilation, but are `NULL` once it has completed. Unfortunately there is no high road to understanding which fields may be modified in a given front end; understanding of the front end's logic and some trial-and-error are needed.

[2] `VIEW_CONVERT_EXPR` was chosen only for convenience; in another front end I would most likely have introduced a new tree code.

3.3 Memory Management

In order to prevent the server's memory use from growing without bound, the server uses a heuristic to decide which trees to keep and which to discard.

Currently the heuristic is based on the notion of a compilation job. A job is a mapping from an object file name to a set of parsed hunks; I chose object rather than source file names because it is not uncommon to compile a source file multiple times in a given project.

When a new job is submitted which has the same object file name as an earlier job, we keep the old job's hunk set while compiling the new job, and discard the old set when the new job is finished. This is done because the old and new jobs typically share a large number of trees.

3.4 Open Problems

There are two open problems with the C front end modifications.

First, the dependency checking for hunk reuse does not take into account the state of pragmas that may be in effect. This can easily be fixed by including some information about pragma state in a hunk's prerequisites.

Second, the move to mapped locations made it possible for the server to run out of location numbers. My current plan is to replace a mapped location in saved tree with an index into a (conceptual) token buffer; then when remapping declarations the location can be replaced with the location of the token at that offset in the new hunk.

4 Results

First, and most importantly, the incremental compiler does not appear to be any slower than the trunk compiler. This is an important result, because it means that the non-decl-smashing front end should still be suitable for general use.

The initial prototype work was all done using `-combine`. I observed large speedups and memory use improvements using the decl-sharing code in this mode; however at the time of writing this is broken, so there are no concrete numbers worth reporting. Nevertheless, this work will improve `-combine` and thus be useful even without incremental code generation.

When running as a server, the incremental C front end is faster than the ordinary front end for some programs. There is a lot of variance here; I speculate that the speedup seen depends on the amount of reuse that is possible, and perhaps even the kinds of hunks seen.

So, for instance, the compile server is approximately 30% faster when compiling Gnome[1] zenity (a small program) and 5% faster when building the Gnome panel. On the other hand, it shows no speedup when building the `src` tree (I used a checkout including `gdb`[4] and `binutils`[2]) or GNU make[3].

Seeing any speedup here is somewhat of a surprise—the C front end is not excessively slow and does not dominate `-ftime-report` output in my experiments. I expect to see better speedups from incremental code generation (for the recompilation case), and from work done on the C++ front end.

Nevertheless, more work here is needed to characterize the improvement. Some interesting avenues are seeing whether the size of the project is important, how much the ratio of system- to user-headers matters, and whether having multiple variants of a given hunk affects the outcome.

The overhead associated with hunks and their prerequisites is not very large. The parsed form of the entire `src` tree results in a `cc1` process that uses 163 megabytes of memory.[3]

5 Future Milestones

5.1 Incremental Code Generation

The server-ized compiler with its incremental front end can easily tell which functions need to be recompiled. If a function's tree was mapped in from an already-parsed hunk, then we have already compiled it. Otherwise, we have never seen it before, and so it must undergo code generation.

Incremental code generation, the second milestone for this project, will take advantage of this fact. Conceptually, this will work by saving the object code for a compiled function, and then reusing this object code as needed.

[3] Just a hair below Emacs—and thus reasonable.

At the time of writing, I have not implemented incremental code generation. I am considering a couple possible approaches:

- Make a single object file for each definition, and relink these object files, using `ld -r`, to produce the object file the user actually asked for.

  This could be done either in gcc directly, or by modifying the assembler to emit multiple object files, split at definition boundaries.

  This approach requires a cache of object files managed by the server.

- Rather than make many small object files, emit an object file containing the modified functions. Then, run the linker in a special mode to stitch together the previous object file and the new one, with definitions in the new object file being preferred.

I'm currently working on a prototype of the second idea, using `objcopy` rather than a linker modification. My initial draft updates the front end to track whether a function must be recompiled, and passes this information to `cgraph`. This yielded an addition 30% speedup when recompiling an unmodified zenity.

Code generation is currently done by calling `fork` in the server process before gimplification. One reason for this is to allow the C front end to re-smash decls and types before gimplification.

5.2 Open Problems in Code Generation

There are two as-yet-unsolved problems in incremental code generation.

First, handling debug info is tricky, because we omit location information from hunk signatures. If a declaration does move, we want to update the debug info, ideally without recompiling the entire function.

Second, because we do code generation in a child process, the compile server does not know what inlining decisions are made, or what changes are likely to make a function become a candidate for inlining. These problems may derail the idea of using ABI compatibility rather than identity comparisons for tree reuse. However, one possible fix would be to provide feedback from the code generator to the server.

5.3 C++

I plan to begin work on the C++ front end after finishing incremental code generation. Given that the C++ compiler generally spends more time in the front end than the C compiler, I expect greater speedups there. I hope to use similar implementation techniques in the C++ compiler as were used in the C compiler; but as C++ is a more complex language, some differences will be needed.

5.4 Incremental Linker

The final deliverable is an incremental linker. It does not do much good to have an incremental compiler if users end up having to wait for a very slow link step. Hopefully `gold`[7] or the `elfutils` linker will solve the problem before I get to this point.

6 Wild Speculation

Currently, the lower bound on incremental compilation is the time needed to preprocess the incoming jobs. This will likely prove to be too slow[4]; in this case, an incremental preprocessor[6] could be written.

I looked into making a multi-threaded compile server, to take advantage of multi-core machines. However, this is somewhat ugly[5] and difficult, would require a stronger guarantee than the "relative immutability" implemented in the current tree sharing approach, and would interact poorly with `fork`. Instead, I intend to add a `-j` option to `gcc --server`, and start multiple compile servers. This is not quite as good as multi-threading, but it is much simpler to implement.

The compile server may be usefully integrated into IDEs—for instance, for analysis or smart completion. This could be easily done via extensions to the client/server protocol.

Forking for code generation may interact poorly with the garbage collector. In particular the child process may dirty many pages unnecessarily. So far I have only been able to picture relatively drastic solutions: do not fork, implement a generational GC, or remove the GC

[4] Considering that the ideal situation would be that insignificant changes are recompiled beneath the user's perceptual threshold.

[5] To the tune of 841 `__thread` variables.

entirely. Removing the need to fork is attractive as it would re-enable investigation into multi-threading; this is the most likely candidate for further investigation.

Any change to reduce the size of a tree node is likely to affect the incremental compiler more than other uses of GCC. I may look into shrinking a node or two.

It may be worthwhile to delete the bodies of non-inlineable functions. As these functions will not be re-compiled, the bodies are not needed after the initial code generation.

The current implementation can only scale to programs that will fit in memory. It might be interesting to reuse machinery from the LTO project to write out hunks, and then only cache the most frequently used hunks in memory.

7 Comparison With Earlier Work

This implementation differs from an earlier attempt at a compile server[5] in a number of ways.

The old server attempted to integrate preprocessing into the model. According to the paper, this was unfinished. The new approach ignores the preprocessor for the time being.

The old server attempted (though again, according to the paper, this was unfinished) to avoid the decl-smashing problem by the use of an undo buffer. The new code separates each instance of a declaration.

The current code lifts a number of restrictions documented in the previous paper:

- All preprocessor-related difficulties are gone, as we ignore the preprocessor.
- The earlier approach documented difficulties with "non-nesting," for instance the use of conditional compilation inside a declaration, or the use of `extern "C"`. There is a similar problem with the new approach but it is easily handled by the hunk boundary computation code.
- In the new code, objects need not be declared in a single location, as we omit the location from the hunk checksum.

8 Conclusion

The incremental compiler project intends to reduce the typical edit-compile-debug cycle by speeding up the compile step in common cases. The compile server is already usable for real programs, and provides a modest speedup in some cases. It may also make `-combine` more practical. Future work promises to deliver even better performance. In addition to the benefits to users, this project tries to provide some benefits to GCC as well, in the form of further separation of the front and back ends, and other general cleanups.

9 Extra

I'd like to thank Red Hat for funding this work and Mark Wielaard for his excellent comments on early drafts of this paper.

The code for this project is all in `svn`, on the `incremental-compiler` branch. There is also a wiki page at `http://gcc.gnu.org/wiki/IncrementalCompiler`.

References

[1] GNOME: The Free Software Desktop Project. `http://www.gnome.org/`.

[2] GNU binutils. `http://sourceware.org/binutils/`.

[3] GNU Make. `http://www.gnu.org/software/make/`.

[4] The GNU Project Debugger. `http://sourceware.org/gdb/`.

[5] Per Bothner. A Gcc Compile Server. In *The GCC Developers' Summit Proceedings*, 2003. `http://per.bothner.com/papers/GccSummit03/gcc-server.pdf`.

[6] Elad Lahav and Andrew J. Malton. Incremental Parsing and the C Preprocessor. `http://www.cs.uwaterloo.ca/~elahav/inc_cpp.pdf`.

[7] Ian Lance Taylor. New ELF Linker. `http://sourceware.org/ml/binutils/2008-03/msg00162.html`.