Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
David Edelsohn, *IBM*
Richard Henderson, *Red Hat, Inc.*
Andrew J. Hutton, *Steamballoon, Inc.*
Janis Johnson, *IBM*
Toshi Morita
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Al Stone, *HP*
Zack Weinberg, *Codesourcery*

Proceedings Formatting Team

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

Inter-Module Analysis in GCC

Geoff Keating
Apple Computer, Inc.
geoffk@apple.com

Abstract

In C, a program is made up of one or more translation units. Historically, GCC has compiled one translation unit at a time, completely discarding state when switching between translation units. It would make interprocedural optimisations more effective if this state could be kept. Inter-Module Analysis (IMA) for C achieves this by extending GCC to work on more than one translation unit. The most significant implementation difficulty was not in implementing the semantics, but in dealing with the problems of scale that appear when very large programs are compiled and the optimisers work on them.

IMA can be extended to work on C++ just as it works on C. Because of the differences between the languages, especially the existence of the One Definition Rule (ODR) in C++, some of the internal implementation details can be streamlined. IMA for C++ allows the possibility of saving compilation time by exploiting the ODR. IMA for C++ also permits an implementation of the `export` keyword.

Although these IMA implementations have some advantages, most notably that they are relatively easy to implement, they do lack some desirable features. They require re-parsing and re-optimisation of every file, even if only one has changed; they do not permit mixing of code from different languages; they do not easily lend themselves to a disk-based compiler. Solving these problems requires some kind of language-independent intermediate form. Unfortunately, at present no-one is working on implementing such a form; volunteers would be appreciated.

1 Introduction

Every compiler is fundamentally limited in the optimisations it can perform by what it knows or can deduce about the code it is optimising. For example, an inliner cannot inline a function whose body is unavailable and constant folding cannot be performed on an expression that refers to a constant whose value is unknown.

One way to improve this is to add special annotations to code and declarations to provide the compiler with information about the functions that it can't see. This is effective, but requires significant work on the part of the user and is error-prone; consider the number of times that a function has been marked `const` when it should not have been. An alternative and often better solution is to give the compiler the ability to see the original definitions of the functions and variables. The compiler can then make deductions from those definitions.

The first step in this direction was unit-at-a-time mode which allowed the compiler to see

the definition of every function in a compilation before compiling the first one. Yet for C and C++ the compiler still examined its input files one at a time; different files were compiled in different processes and there was no possibility of information sharing.

2 IMA for C

In early 2003 GCC was changed so that for C it can share information between the compilation of all the files given on its command-line, to permit Inter-Module Analysis (IMA). There were three key parts to the change: the driver changes, the C semantic changes, and the cross-module linkage step.

2.1 Driver Changes

The highest-level change was to the driver. GCC has always been able to accept

```
gcc file1.c file2.c -o ex
```

but previously the driver would convert this into a sequence of executions like

```
cc1 file1.c -o file1.s
as file1.s -o file1.o
cc1 file2.c -o file2.s
as file2.s -o file2.o
ld file1.o file2.o -o ex
```

This does not permit any information sharing between the two `cc1` invocations. After the change, the driver now passes the two source files to the same `cc1` invocation, like

```
cc1 file1.c file2.c -o ex.s
as ex.s -o ex.o
ld ex.o -o executable
```

The driver was also changed to permit generation of a `.o` file (or a `.s` file) from more than one source file; where before it only allowed

```
gcc -c file1.c -o file1.o
```

it now permits

```
gcc -c file1.c file2.c -o ex.o
```

This last point was the cause of some controversy, as some argued that it should be

```
gcc -c file1.c -o file1.o \
  file2.c -o file2.o
```

however, I believe that this would have been difficult, dangerous, and useless. Part of the difficulty would come from the need to modify the high-level structure of `cc1` to make it output more than one `.s` file. Further difficulties occur when you consider that now it can happen, through inlining, that `file2.o` might need to be able to refer to `static` variables in `file1.o`.

Consideration of such a case will then quickly lead to the realisation that the separation of the `.o` files is not useful, since there is no situation where one could reliably appear without the other; they must be rebuilt at the same time because they both depend on both `file1.c` and `file2.c` and they must be linked together into every executable or not at all. In fact, if you wished this effect you could simply have written

```
gcc -c file1.c file2.c \
  -o file1.o
gcc -c -x c /dev/null \
  -o file2.o
```

The danger appears when you consider that even though the two `.o` files must be used together, if a true separation was implemented where an attempt was made to output functions from each `.c` file into the corresponding `.o` file, those dependencies might not be apparent. Users might be led to think that they could perhaps use or rebuild one `.o` without the other, causing inconsistent behaviour of the built executable or link failures that appear only at certain optimisation levels. Thus, this feature is not implemented in the driver.

2.2 C Semantic Changes

Unlike some languages, in C one cannot simply concatenate two source files and obtain the same behaviour as if they had been compiled separately. C defines a *translation unit* to be the result of preprocessing a single source file and says that all declarations have a *scope*, a region of the program text in which they are visible. In particular, any declaration which is not inside some other construct has *file scope*, which terminates at the end of the translation unit.

Since the compiler had not previously needed to consider multiple translation units, it had placed such declarations in a single topmost scope which it never needed to terminate. It did have existing support for nested scopes and even for handling both variables (as part of ISO C) and functions (as a GNU extension) in them, so the work required to have multiple topmost scopes was small.

The C frontend used the existing support for nested scopes to ensure that `static` objects with the same name in different input files were named uniquely in the assembly output. This was achieved by setting the `DECL_CONTEXT` field.

Finally, there was additional code required to have `comptypes` correctly implement the ISO C semantics [2, section 6.2.7 paragraph 1] for type compatibility across translation units. Within a single translation unit, every structure, union, or enumeration definition creates a new type which is incompatible with all other types, but between translation units, structures are compatible if they are sufficiently similar.

2.3 Cross-module Linkage

The third key component of IMA for C is the logic which associates a declaration in one translation unit with its definition in a different translation unit.

In the original implementation, a final linking phase was implemented in `merge_translation_unit_decls`. Once all the source files have been parsed into `tree`s, all the globally visible definitions were merged with their declarations from other source files, just as a linker would associate references with definitions.

At the time, this was necessary because GCC permitted, apparently as an extension, a declaration to be declared first `extern` and then later `static`. Because of this, no declaration could be certain to be accessible outside its translation unit until all of that translation unit was seen. It was not clear that the extension was intentional and not simply a bug. When it caused further trouble, it was removed for GCC 4.0.

The current compiler implements the association by creating only a single declaration for each object and using it consistently. The type of the object will be whatever type it was last declared with, except that for arrays of unspecified length, like `extern int foo[];`, the type may contain the length if one was seen earlier. A consequence of changing the type of the declaration like this is that although the type

of an object itself will be complete, the types of objects pointed to from the type may not be complete even if they were complete when the object was originally declared; this is believed to not affect compilation.

An interesting feature of the C language is that type compatibility is not transitive. That is, just because type *A* is compatible with *B*, and *B* is compatible with *C*, it doesn't follow that *A* is compatible with *C*. This can happen if *A* and *C* are structures, unions, or enums in the same translation unit and *B* is in a different translation unit. Combined with the modification of types of objects, this means that type checking is not reliable after all the source files have been turned into `tree`s. Fortunately, at that point full ISO C type checking should not be necessary.

Another case where type compatibility is not transitive is where *A* and *C* are complete structure or union definitions in different translation units with the same tag but with different fields and *B* is an incomplete structure or union declaration in a third translation unit. In the 1989 version of ISO C, such cases are much more common because that version does not even require that the tags match [1, section 6.1.2.6]. Current GCC does not handle some rare consequences of this case properly in the tree-level optimisers.

2.4 Implementation Experience

All the above caused relatively little trouble in the initial implementation. The greatest problems were caused by the compiler using all the extra information available in inlining, then performing subsequent optimisations to produce procedures that are larger and have a significantly different structure than any routine that a human (or even most programs) would write. These routines tend to have a more complex loop structure with much deeper loop nesting, many more variables, longer basic blocks, and more basic blocks. All this stresses the existing optimisers significantly, exposing semantic and performance bugs.

The effect of C IMA was significant. The initial implementation, with no other tuning, provided an improvement of about 5% on SPEC2000. Since then, further improvements have been made in the optimisers leading to an even greater performance impact.

3 IMA for C++

Having IMA for C is useful, but much code today is written in C++. At present, there is no IMA functionality for C++, but let's consider how it would be implemented.

Much of the implementation would follow the C version. The same driver changes would be used, but there would be some significant differences in semantic processing. In ISO C++ [3], types have *linkage*, just as functions and variables do in both languages and, instead of the requirement for *compatible* types, the requirement is simply that the types of objects are "identical" [3, `basic.link` paragraph 10] after some basic simplification.

Thus, for C++, there should be no need to change the types of declarations and all the discussion above about the consequences of intransitivity does not apply to C++. Instead it will be necessary to track whether those objects that must be defined, rather than declared, before being used actually have been defined in this translation unit.

3.1 One Definition Rule

While considering objects that can be defined in more than one translation unit, there is another feature of ISO C++ that GCC could benefit from, the One Definition Rule (ODR) [3, `basic.def.odr` paragraph 5]. Part of the ODR is that when multiple definitions are allowed, each definition must contain the same sequence of tokens and those tokens must have the same meaning in every translation unit.

This means that it is not necessary to completely parse an entity every time that it is seen. A loose implementation could simply notice that the entity has been seen before and is being defined here and then skip to the end of the entity. A better implementation would check that the token sequence is really the same and an even better one would also check that the meanings are the same. For backwards compatibility, GCC will probably need to be able to do both the 'loose' and 'better' versions, since it is likely that many existing programs rely on the existing lax behaviour.

An additional benefit of compiling multiple translation units to a single `.o` is that the compiler will need to emit only a single copy of implicit template instantiations and out-of-line copies of inline functions. The effect of these benefits should be that compiling two or more files together should be significantly faster than compiling them separately.

3.2 Templates

If all was as it should be, handling template instantiation with IMA would not be a concern. Unfortunately, GCC's handling of templates is not quite right at present [4]. The ISO C++ standard specifies precisely when a template should be instantiated. It is possible to construct examples to test this by having incomplete structures completed after that point or overloading functions after that point. GCC presently instantiates templates at the end of the compilation, not at the point specified by the ISO C++ standard.

For simple IMA, this could just be changed to have GCC instantiate templates after each compilation unit, which would preserve the existing incorrect behaviour. For proper ISO C++ behaviour, especially for `export` (see below), it would be best if template instantiation was changed to operate at the point ISO C++ requires, at least from the user's viewpoint.

3.3 `export`

There is one significant ISO C++ feature that GCC does not yet implement, the `export` functionality. For those readers unfamiliar with this (perhaps because almost no existing compilers implement it), this feature allows the user to define a template in one translation unit with the `export` keyword. In other translation units the template need only be declared; the compiler will use the definition that had the keyword.

Once GCC has the ability to process multiple C++ translation units at once and the template instantiation problem is fixed, then the only step required to implement `export` is to link the declaration with the `export`ed definition. GCC would do just what it would do with any other kind of declaration while suppressing the error that would have been produced if the `export` keyword was not there.

4 Multi-Language IMA

The work described in the previous sections allows the compiler to combine information from

multiple files, but only so long as those files are written in the same language. Programs are commonly written in multiple languages so it would be desirable to propagate information across language boundaries.

One way to do this would be to merge all GCC's supported languages into a single executable and extend that so that each source file compiled could be a different language. It would be desirable to share front-end code between the languages as much as possible, maybe even going so far as to share one parser between C and C++, but that is not required for this functionality. Such a merge would be a worthy project for many reasons, not least of which is that it would mean that the interface between language frontends and the middle-end would become better-defined. Such a merge would also be a lot of work.

Another disadvantage of the IMA implementations described earlier is that even if only a single file is changed, all files must be reparsed. The repeated work makes this very slow when compared to compiling that single file by itself. If some of that work could be avoided then there would be a significant speedup. The IMA compile would still be many times slower than compiling that single file separately because all the optimisation and code generation would still have to be re-done unless some more sophisticated dependency analysis was performed.

The proposed design shown in Figure 1 is one way that these problems can be avoided. The compiler has a number of parsers, just as it does now (possibly in the same executable, possibly not).

The language frontends call into an 'Intermediate Language (IL) library' in much the same way as they now call into the middle-end of GCC. The interface would be similar to the current GENERIC, although hopefully a smaller, more efficient GENERIC than the one we have today.

The IL library creates an IL representation in SSA form suitable for optimisation. This intermediate form can either be sent directly to a very fast code generator that would run at `-O0` or sent through one or more simple optimisers, including at least dead code removal and maybe CSE and other optimisations that simplify the code. The aim here is not to generate great code, but to reduce the amount of work that later passes must do.

Next, the IL is written out to an on-disk format together with whatever analysis information makes the job of the remaining passes easier (trading off a little disk space against faster compilation). The on-disk format can be directly executed, possibly even JIT compiled if that turns out to be faster than the `-O0` code generator.

Usually, though, the on-disk format will be passed to what we would now consider to be a GCC frontend for the IL 'language'. The new frontend will be able to read multiple IL files and merge them; that is, it will support 'IMA for IL'. It may also be desirable, in the case of a single input IL file, to skip the writing and reading and just pass the data in memory. If it is on disk, the IL will be not be read all at once, but as needed for inlining or to output a particular routine.

Each routine, or possibly a group of routines that are being optimised as a unit, will then be passed to the later single-routine optimisers and through RTL-based code generation.

This design should give better compile speed performance and work in restricted (32-bit) address spaces while supporting IMA and the design is similar to well-tested designs that other compilers use. Unfortunately, there are a lot of pieces which would have to be written or

re-written to implement this solution, most notably the IL format must be defined and the reader and writer libraries must be written.

5 State of This Work

IMA for C is in current compiler releases; credit is due to Caroline Tice and Per Bothner for the driver changes and to Zack Weinberg for the changes to use a single declaration for each object. IMA for C++ is planned and will be done when time permits, earlier if a volunteer appears, or not at all if the multi-language IMA is ready before IMA for C++ can be started (since the multi-language IMA would make the single-language IMA implementations obsolete).

The multi-language IMA is not planned and at present won't be done until a volunteer (or a group of volunteers) steps forward to do the work. It may be that there is no strong need for multi-language IMA, in which case it need not ever be implemented. However, I believe it would be useful and so I take this opportunity to encourage like-minded people to come forward to help on this project!

References

[1] International Organization for Standardization. *ISO/IEC 9899:1990: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 1990.

[2] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, December 1999.

[3] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, October 2003.

[4] Geoff Keating. GCC Bugzilla bug c++/16635.

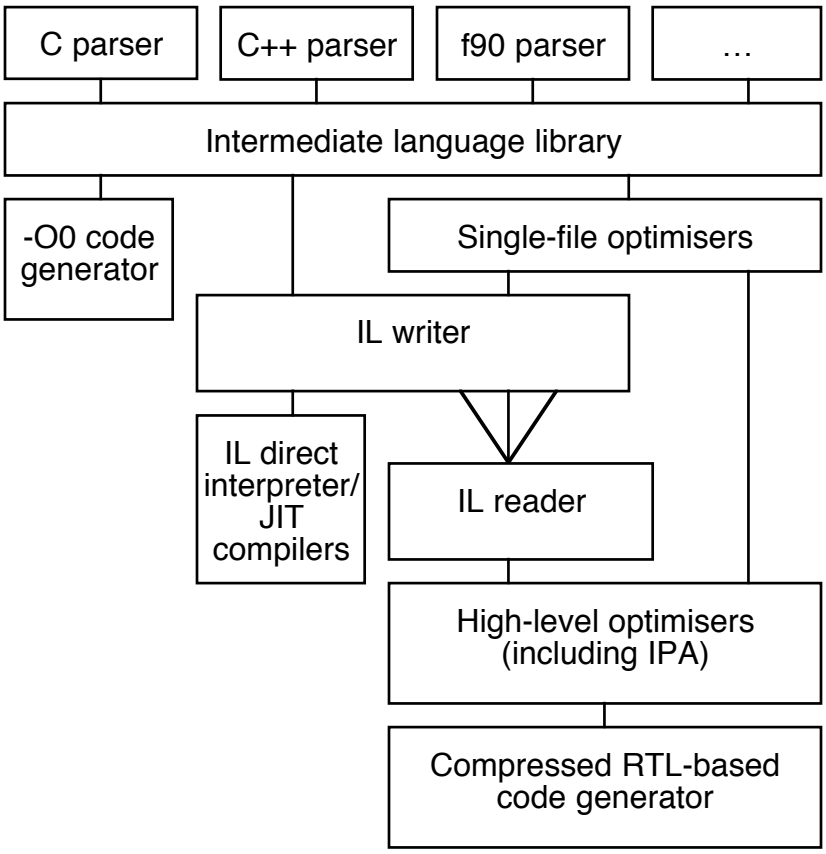

Figure 1: Multi-Language IMA Design