# Extending GCC with Modular GIMPLE Optimizations

Sean Callanan, Daniel J. Dean, and Erez Zadok
*Stony Brook University*

**Appears in the proceedings of the 2007 GCC Developers Summit**

## Abstract

We present a system of plug-ins for GCC that allows GCC to load GIMPLE transformations at run-time. This system reduces the support effort required for GCC by separating transformations from the core compiler. It also makes it possible for developers not connected with the GCC project to develop and distribute transformations independently. We demonstrate two plug-ins we have developed with this system, one of which reduces the effort required to develop transformations significantly by allowing visualization of the GCC control-flow graph and the GIMPLE tree structure. We enumerate portions of the compiler that could be extracted into plug-ins, and describe future applications of the plug-in system.

## 1 Introduction

GCC is considered the reference compiler for real-world C code. It is provided with all of the popular commercial and free Unix systems, making its availability nearly ubiquitous. Virtually all modern C code-bases are compatible with GCC, making it general. Finally, some code-bases, such as the Linux source code, explicitly rely upon features of GCC in order to work properly, making GCC's implementation of the C language something approaching a de facto standard.

This makes GCC an attractive platform for production-grade code transformations. Code transformations are compilation passes that modify source code in some way that is nonessential to its translation to machine code. They can be used for optimization, for instrumentation, and to make cross-cutting modifications to aspects of the code being compiled, among other applications. These transformations range from frequently-used optimizations like function inlining to special-purpose debugging transformations like Mudflap [2].

GCC is even more attractive for transformation development because of the GIMPLE intermediate representation [6], which provides a stable, easy-to-use API for inspection and manipulation of intermediate code. The use of the GIMPLE intermediate representation allows high-level optimizations to avoid having to use the RTL (Register Transfer Language) layer, which previously made GCC transformation a more difficult task. Before GIMPLE, research compilers or alternative approaches like CIL [4] were more attractive, but these have their own problems, particularly the necessity to maintain compatibility with GCC.

However, the GIMPLE intermediate representation presents its own challenges during transformation development and testing for several reasons. First, the maturity of the GCC project and the fact that many system distributions depend on GCC to compile their system makes it difficult to get transformations integrated into GCC until they are very mature. Second, it may not be desirable to include and maintain transformations that do not have broad appeal in the core GCC distribution. Finally, it is an unattractive proposition to have to distribute experimental transformations as patches against a particular version of GCC and recompile the compiler when changes are made.

To solve these problems, we developed a plug-in system similar to that used by Eclipse [5]. Our system allows separate development and compilation of GIMPLE transformations, solving the problems listed above and offering new features like enhanced debuggability and better argument passing. We have already developed a variety of plug-ins using our system, and have realized two main benefits. First, we were able to take advantage of graphical debugging tools that we describe in Section 3.2 as well as significantly reduced development time because we were developing outside the GCC build system. Second, we were able to port our transformations from one version of GCC to another without changing a *single* line of code; once the plug-in support was ported to the new GCC release, the plug-ins just needed to be recompiled.

The plug-ins we have developed are under the GPL, and we anticipate the possibility of enforcing the GPL on all GCC plug-ins by requiring plug-ins to export a function `is_GPL` which returns 1, analogously to the Linux kernel's taint mechanism. Depending on GCC developer policy, returning 1 could be made mandatory in order for GCC to run the plug-in.

In this paper, we demonstrate the simplicity and power of GCC transformation plug-ins. In Section 2, we describe the modifications to GCC that make plug-in–based development possible. In Section 3, we describe some plug-ins that we have already built using this infrastructure, highlighting plug-ins that are useful to transformation developers. In Section 4, we discuss two parts of GCC that could be made into plug-ins. In Section 5, we describe plug-ins that could be created in the future, and we conclude in Section 6.

## 2 Modifications to GCC

Plug-ins are built based on an Autoconf-based template [1]. The template's `configure` script currently requires the headers from a built version of the GCC source code; when the plug-in is built, the Makefiles produce a shared object file suitable for loading using the host operating system's dynamic loader interface.

Only minor changes need to be made to GCC to support plug-in loading. These changes revolve around three tasks; we will discuss them below in turn. The first change is an addition to the GCC build sequence, compiling the Libtool `ltdl` library [3] into GCC and linking GCC with `-export-dynamic`. This allows GCC to load plug-ins, and allows plug-ins to access GCC interfaces. The second change is the addition of an optimization pass before all other GIMPLE transformations, and at the start and end of translation for each file. This allows plug-ins to maintain per-file state and perform code optimizations while referring to this state. The third change is the addition of a compiler flag that allows the user to specify plug-ins to load and provide arguments to those plug-ins either on the command line or through files.

To add the `ltdl` library to GCC, we modified the top-level top-level Makefile to add build rules for the `ltdl` library. Additionally, we modified the build rules for the `cc1` binary to make it compile with Libtool, export its symbols like a shared library (using the `-export-dynamic` option to Libtool), and use the `ltdl` library to load plug-ins. The ability to export symbols from an executable to plug-ins does not exist on every platform: Linux, Solaris, and Mac OS X support this functionality, for instance, but Cygwin does not. A build process in which the GCC backend code is linked as a shared library, and `cc1` and all plug-ins are linked against it, would have eliminated this requirement. However, large amounts of state that is currently maintained as globals by the backend would have to be converted to on-stack state because otherwise `cc1` and the plug-in would have differing copies of the backend's global state.

To allow instrumentation plug-ins to run at the proper times, we added a GIMPLE transformation pass that occurs before `pass_all_optimizations` in `passes.c`. This allows plug-ins to act on unoptimized code, before inlining has occurred. We did this here because many plug-ins we have developed are for debugging purposes, and require as pristine a view of the original code as possible. Additionally, we provided hooks at the beginning and end of translation to allow plug-ins to initialize and clean up internal state. We anticipate that, in the future, GCC developers will add hooks for other GCC optimization passes, such as the C GENERIC transformation phase and the RTL optimization phase, and other points in the GIMPLE compilation phase.

Finally, to allow the end user to specify which plug-ins should be loaded with which arguments, we provided a new argument, `-ftree-plugin`, which has the syntax shown in Figure 1.

`-ftree-plugin=`*plug-in-name*<br>
  `:`*key*=*value*<br>
  `:...`

*Figure 1: Syntax for specifying a plug-in.*

The first argument, *plug-in-name*, is a shared object file that contains the functions `run`, `pre_translation_unit`, and `post_translation_unit`. The `run` function is called for each function in the translation unit; the other functions are called before and after the entire translation unit is processed, respectively. The list of *key-value* pairs specifies arguments to the plug-in; these can be fetched using a function. In addition, the special key `_CONF` specifies a file to be loaded and parsed for additional arguments; in this case, each line in the file is a key-value pair separated by an = sign.

## 3 Existing plug-ins

In this section we enumerate some plug-ins that we have already developed. In Section 3.1 we discuss a verbose dump plug-in for GIMPLE meant for use by programmers in developing transformations, and in Section 3.3 we describe a call-trace plug-in for use by end users in tracing their code. We have also developed `malloc` checking and bounds-checking plug-ins; however, these will be superseded by a plug-in implementation of Mudflap (see Section 5).

### 3.1 Verbose Dump Plug-in

Transformation developers frequently require a view of the GIMPLE code that is as verbose as possible. They use this view for several purposes: to identify patterns that need to be transformed, to determine the proper form of GIMPLE structures that transformations should generate, and to verify that transformations are working correctly. We designed a verbose dump plug-in to facilitate this. We designed the verbose dump plug-in with extensibility in mind: as GIMPLE evolves and grows, the verbose dump plug-in will handle new GIMPLE objects, such as new tree codes or parameters, with little or no changes needing to be made. We achieved this by creating a new file, `parameter.def`, that resembles `tree.def` but formally specifies all the accessor macros that exist for tree attributes. The file contains lines of the form shown in Figure 2.

```
DEFTREEPARAMETER(
    name,
    type,
    macro,
    code, ...
)
```

*Figure 2: Syntax of* `parameter.def`

The *name* field specifies the name of the macro; the *type* field specifies what type of data it returns (e.g., `SIZE_T` or `TREE`); the *macro* field specifies the macro used to extract the field; and the *code* fields constitute a list of `TREE_CODE`s for trees that have this parameter. For example, the parameter named `type_precision` has type `SIZE_T`, macro `TYPE_PRECISION`, and codes `INTEGER_TYPE`, `REAL_TYPE`, and `VECTOR_TYPE`.

## 3.2 Graphical Inspection of GIMPLE Code

As shown in Figure 3, the output from the verbose-dump plug-in is so verbose as to be overwhelming in large quantities. Rather than adopt a simplified representation, we instead developed a Java-based tool called Gimple Viz to represent the output graphically. We chose Java as the development language due to its cross-platform compatibility, which allowed us to concentrate on the development of the actual tool itself as opposed to platform support and library dependencies. Figure 4 is a screenshot of Gimple Viz displaying a file. The visualizer has three main areas: the Control Flow Graph area, the GIMPLE Tree View area, and the Source/Search area, which we describe below.

**Control Flow Graph: .** The control flow graph for each function is rendered as rectangles connected by arrows. Each colored rectangle represents a basic block. When the user clicks on a block, Gimple Viz highlights the selected block along with its predecessors and successors. The successor edges are highlighted as well. Additionally, it displays a tree representation of the corresponding GIMPLE nodes in the GIMPLE tree view area, and highlights corresponding code or dump lines in the Source/Search area.

**GIMPLE Tree View: .** The GIMPLE tree view area is a visual representation of the GIMPLE code for a particular basic block. The root node of each tree is a statement from the currently selected basic block, labeled with the result of applying `print_generic_stmt`. The other nodes are operands or parameters of their parents. The user interacts with the tree view in two ways: clicking and searching. Manually clicking a node will expand that node showing its children. This process can be repeated until the desired node is reached. Searching for a particular `TREE_CODE` will expand the tree to reveal the desired node, allowing the user to quickly locate specific nodes.

**Source/Search: .** The Source/Search area can show search results, source code, and verbose-dump output. The results of searches—function searches, basic-block searches, and type searches—are interactive: clicking on a function search result shows the control-flow graph for that function; clicking on a basic-block search result shows the containing function's CFG and highlights the block; and clicking on a `TREE_CODE` search highlights

```
MODIFY_EXPR 1,2
 TREE_TYPE:
   INTEGER_TYPE 2,0
     TYPE_PRECISION=32
    TYPE_UNSIGNED=true
 VAR_DECL 2,0
  TREE_TYPE:
    INTEGER_TYPE 2,0
      TYPE_PRECISION=32
      TYPE_UNSIGNED=true
  DECL_ARTIFICIAL=true
 MULT_EXPR 1,2
  TREE_TYPE:
    INTEGER_TYPE 2,0
      TYPE_PRECISION=32
      TYPE_UNSIGNED=true
```

*Figure 3: A portion verbose dump output for one statement, leaving many node attributes out.*

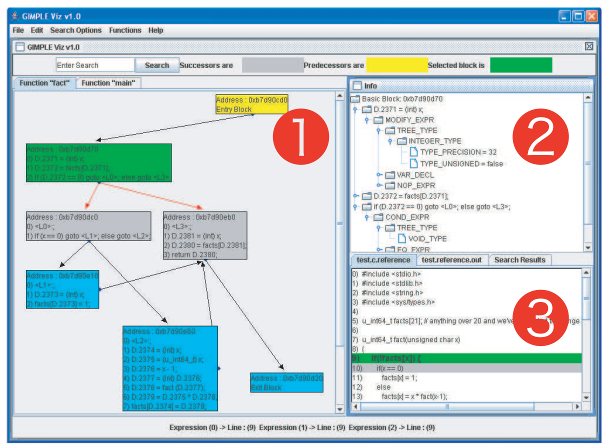


*Figure 4: Gimple Viz displaying a file. 1 marks the CFG area, 2 marks the tree view, and 3 marks the source/search area.*

```
*** CALL TO main [0]
Struct :**test** found in function
**main**
test->value = (int32_t)5
test->name = (char*)"contents"
** CALL TO foo [1]
* [1] testPtr = 0x0x7fffcef31770
*** CALL TO foo2 [2]
* [2] value = 0x0x7fffcef31748
Conditional found value = FALSE,
right branch taken...
*** [2] RETURNED null
```

*Figure 5: Call trace output*

the containing basic block and expands the containing tree in the GIMPLE tree view to make the tree with that code visible.

Gimple Viz can also display the original source file that was compiled by GCC in the source/search window. For quick reference, line numbers are displayed for the user. Although the user cannot directly interact with this area, clicking a basic block or a search result will highlight the lines corresponding to that block, its predecessors and its successors. Finally, the source/search window can also display the raw verbose dump output.

## 3.3 Call Trace Plug-in

We have developed a plug-in called *call-trace* to allow full verbose tracing statements to be added to a program at compile time without requiring the programmer to add any code. This feature significantly reduces debugging time for many code problems by eliminating the need to add `printf` statements and other debugging statements to code, and by providing verbose tracing information in cases where the programmer would normally have needed to single-step the program in `gdb`.

This plug-in identifies control points in the GIMPLE code corresponding to conditional statements and function calls, as well as accesses to variables. Arguments control exactly which statements are logged, and which portions of the source code are to have logging added. The way events are reported is also configurable: logging statements can be printed using `fprintf` or sent to a custom logging function. Figure 5 shows sample output from the call tracer.

We are currently developing an extension to Gimple Viz to display the output from the call-trace plug-in in a visual manner, giving the developer the ability to watch the internal execution of a program at run-time. We are also expanding the call-trace plug-in to detect not only conditionals but loops as well by tying into the GENERIC intermediate representation.

# 4 Making Plug-ins from Existing Functionality

In this section, we describe portions of GCC's functionality that could be extracted into separate modules for use only when needed. This would have three benefits: first, it would enforce modularity for these components, ensuring that they can be maintained separately from the main code base and contributing to their stability as GCC internals change. Second, it would reduce the turnaround time for fixes to mudflap because they would not need to be subject to the scrutiny that core GCC patches are subjected to. Third, it would reduce the size of the core GCC code base, resulting in less code for GCC's core developers to maintain and support, and less download and compilation time for end-users.

**Mudflap: .** This utility provides pointer-debugging functionality including buffer overflow detection, matching-based leak detection, and reads to uninitialized objects. It is implemented as two GIMPLE optimization passes: one that executes before the lowering to SSA (Static Single Assignment) so that all scopes are intact, and one that executes after lowering and optimization to get an accurate view of just those memory accesses that have actually been performed. Mudflap can be converted to a plug-in provided that plug-in hooks are provided at multiple stages in the optimization process. Our plug-in infrastructure supports transformation hooks at all locations where built-in GIMPLE transformations can take place, making this process straightforward.

**gcov and gprof: .** These utilities consume call-graph information that is generated by GCC and by the running program, creating runtime profiles of the execution patterns for code that has been compiled with the `-p` or `-fprofile-arcs` flags. When profiling, GCC modifies the program to include *coverage counters* embedded in the program that provide runtime coverage information. It also generates a call-graph for the program. The transformation that performs these tasks runs as a transformation in a way analogous to Mudflap, but labels basic block edges with additional information that uses the `aux` field in the basic block structure. This does not present a problem for these transformations, since they take place in one pass and do not need persistent `aux` storage. However, other plug-ins that may need to do analyses at multiple times in compilation it may become desirable to expand `aux` to support addition of custom fields, perhaps keyed on a string, at runtime.

# 5 Future Work

Once the groundwork is in place that allows GCC transformations to be developed as plug-ins, we anticipate that many new transformations will be developed. In this section, we outline future applications of plug-ins, some of which we are currently developing for our own research.

**Transformations in Python: .** Some developers only want to perform straightforward analyses or transformations that use the GIMPLE API. To reduce development time for these developers, we are developing a plug-in that will expose the GIMPLE API to Python scripts. This plug-in links against the Python library and executes a user-specified Python script for each function being translated. It currently allows read-only access to basic blocks and trees; we are adding support for viewing and editing the control-flow graph, adding and removing statements, and modifying trees. In addition to reducing development time, this plug-in will allow developers to use Python data structures, reducing implementation time for optimizations that use sophisticated algorithms to perform static analyses on GIMPLE.

**Library call error detection: .** When developing systems software, programmers frequently add large amounts of error checking for library function calls to detect problems that are ironed out in the early stages of development. This error-checking adds to code size, reduces code readability, and takes time. In addition, retroactively adding error-checking onto existing code if it fails can be a significant time investment. . A GIMPLE transformation plug-in could be used to add error-checking to code at compile time, optionally warning when the code is not written to check the result of calls that commonly fail.

**Interface profiling: .** Threaded applications typically have points at which threads wait for responses from other threads. These can take several forms: functions that are called to perform synchronous requests, or locks that the programs block on until data is ready. Additionally, even single-process applications can spend time waiting for library functions or system calls to complete. A GIMPLE transformation plug-in could accept a list of locks and interface functions to profile, and add entry-exit profiling to these locks and functions. This would be coupled to a runtime library that determines the amount of time spent waiting for these interfaces, credited to the functions that waited for them.

# 6 Conclusion

We have described a framework that allows GCC to load and execute plug-ins that implement custom GIMPLE transformations. This framework offers three compelling benefits:

- it reduces development time for new GCC transformations;
- it allows transformations to be developed and distributed that would otherwise be difficult to use or not available at all; and
- it reduces the workload for the GCC core developers by reducing GCC code size and allowing many transformations to be maintained separately.

We have shown a verbose-dump plug-in and a compatible Java-based visualizer that help GCC developers develop and debug their transformations. We have also shown a call-trace plug-in that tracks function calls, variable accesses, and conditionals, providing a detailed view of the execution of a program. In addition to these existing plug-ins, we have shown examples of existing functionality in GCC that could be converted to plug-ins, and examples of new functionality that do not exist yet but would be well-suited to implementation as plug-ins.

We believe that GPLed GCC plug-ins will be a major democratizing force, bringing new developers to GCC and extending the benefits of compiler integration to a wide range of new applications. We are currently developing a set of patches to integrate our plug-in support into the next release of GCC.

# 7 Acknowledgments

This work was partially made possible thanks to a Computer Systems Research NSF award (CNS-0509230) and an NSF CAREER award in the Next Generation Software program (CNS-0133589). We also thank Michael Gorbovitski, Radu Grosu, Annie Liu, Justin Seyster, Scott Smolka, Scott Stoller, and Paul Talamo from Stony Brook University for their advice and feedback developing this system, and Eric Christopher from the GCC project for assistance porting it to mainline GCC.

# References

[1] David MacKenzie and Ben Elliston and Akim Demaille. Autoconf: Creating automatic configuration scripts. `http://www.gnu.org/software/autoconf/manual/autoconf.pdf`, 2006.

[2] F. C. Eigler. Mudflap: Pointer Use Checking for C/C++. In *Proceedings of the First Annual GCC Developers' Summit*, pages 57–70, Ottawa, Canada, May 2003.

[3] Free Software Foundation. Shared library support for gnu. `http://www.gnu.org/software/libtool/manual.html`, 2005.

[4] G. C. Necula and S. McPeak and S. P. Rahul and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, England, 2002. Springer-Verlag.

[5] M. Boshernitsan and S. L. Graham. Interactive transformation of Java programs in Eclipse. In *Proceedings of the 28th Internation Conference on Software Engineering*, pages 791–794, New York, NY, 2006. ACM Press.

[6] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *GCC Developers Summit*, 2003.