The Visual Development of GCC Plug-ins with GDE

Daniel J. Dean
Stony Brook University
`kesmier84@gmail.com`

Sean Callanah
Stony Brook University
`spyfee@gmail.com`

Erez Zadok *Stony Brook University*
`ezk@cs.sunysb.edu`

Appears in the Proceedings of the 2009 GCC Developers' Summit

Abstract

Being able to directly affect code compilation with code transformations allows the seamless addition of custom optimizations and specialized functionality to code at compile time. Traditionally, this has only been possible by directly modifying compiler source code: a very difficult task. Using GCC plug-ins, developers can directly affect code compilation, without actually modifying the source code of GCC. While this makes applying a completed plug-in easy, plug-in development is transformation development nonetheless: an arduous task. The plug-in developer is required to have the same thorough understanding of compiler internals, complex compiler internal representations, and non-trivial source to internal representation mappings as any other transformation developer.

Recently, simplified representations, such as CIL, have been developed to help developers overcome some transformation design challenges. Although useful in their own respect, representations like CIL are often language specific by design. This requires the developer to make the unfortunate choice between relative ease of development on a simplified representation or language generality on a more complex representation.

We have developed a visual approach to transformation development consisting of a two components: a plug-in to extract GCC's intermediate representation and a Java-based tool to visualize it. This thesis demonstrates how our visual technique significantly reduces many of the problems facing transformation development without sacrificing the inherent benefits of a more generalized intermediate representation.

1 Introduction

Developers have long wanted greater control over compilation in order to automatically add features like application-specific custom optimizations, integrated type checking, function call logging, or parallism to code at compile time [2] [30] [24] [21]. Code transformations give developers this ability by modifying the compiler's internal representation of compiling code. The traditional development of code transformations,however, requires the direct modification of compiler source files, a difficult and error prone task. As Section 3 explains, GNU Compiler Collection (GCC) plug-ins are code transformations which do not require the developer modify the compiler source itself [6]. Although this makes the application and deployment of completed transformations a relatively simple process, plug-in development is an arduous task.

The GCC developer community has a great deal of expertise in developing code transformations due to their intimate knowledge of the compiler. Non-GCC developers, however, must first learn the inner workings of GCC before developing a transformation. One of the most daunting tasks in understanding the inner workings of GCC is understanding the various intermediate representations that GCC creates. As shown in Figure 1, a single line of C code produces many GIMPLE trees, with each GIMPLE tree containing internal information. Although each GIMPLE tree node is used by the compiler in one way or another, a typical transformation is only interested in a subset of nodes. Unfortunately, for the developer this often leads to hours of sorting through low-level intermediate code to find a needle in the vast intermediate-representation haystack.

This paper presents a visualization technique for the development of GCC plug-ins. Our technique involves the design and implementation of a visualization tool, the *GIMPLE Development Environment* (GDE), along with a GCC plug-in to extract and format GCC internal informations. GDE provides developers with four types of visualizations: (1) the control flow graph, (2) the call graph, (3) the GIMPLE trees, and (4) the mapping from source to internal representation. We demonstrate with a series of use cases, how these visual representations significantly reduce the difficulty of interpreting and understanding the intermediate representation that GCC generates while compiling a program.

The remainder of this paper is organized as follows. Section 2 gives an overview of GCC as a whole by presenting the fundamentals of GCC. It is here we introduce the various phases of compilation, explain why each phase exists, and finally describe the intermediate representation at each phase. Although each phase is useful in its own right, this paper focuses primarily on the GIMPLE intermediate representation. Next, in order to understand the our visualization technique, we must understand the GCC plug-in system, which we discuss in Section 3. We then briefly explain the development and features of GDE in Section 4.

Once GDE and GCC plug-ins are understood as a whole, we explain how GDE allows for the effective design and debugging of compiler transformations in Section 6. Here we show how we have used GDE in the past to design and debug our own transformations, describing each case in detail along with the specific advantages GDE brings to the development process. We then examine, in Section 5, the DB-dump output of several applications, suggesting analysis that can be done on these dumps. We then further illustrate exactly why GDE was developed by examining some related technologies in Section 7. We conclude in Section 8 by summing up the key points of this paper and finally, discuss further expansion possibilities for GDE in Section 9.

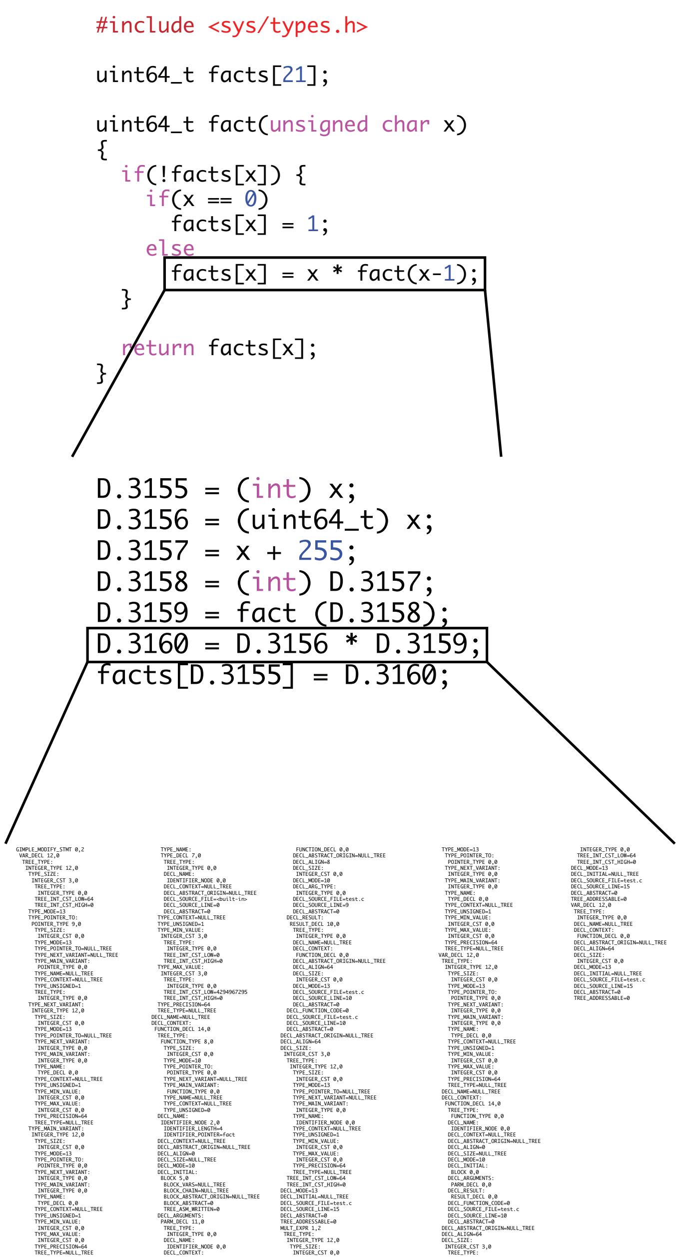


Figure 1: An example showing the C to intermediate representation blow-up.

2 Background

The GNU Compiler Collection (GCC) [13] is an open source compiler which was initially released in 1987 as a C compiler under the name GNU C Compiler. Although initially a compiler only able to compile C code, GCC is now a massive compiler suite able to compile many programming languages, such as C++, FORTRAN, Pascal, Objective-C, Java, and Ada to a variety of architectures. Due to the large number of distinct architectures and languages supported, GCC designers have separated the GCC compilation process into three distinct phases, as seen in Figure 2: the front-end, the middle-end, and the back-end [14]. We discuss these phases next.

2.1 Front-End

GCC's front-end is the language-dependent portion of compilation which is responsible for converting a preprocessed source file into a representation suitable for further compilation. Specifically, the front-end first parses the source code, constructing type and symbol information for compilation. This phase is responsible for operations such as the enforcement of language-level standards compliance, resolution of type definitions, type inference, and construction of scopes. The front end then produces a tree-like intermediate representation, which differs from language to language, while also populating some global variables holding auxiliary information such as the `TREE_ADDRESSABLE` flag, which indicates an item can be passed to the runtime. This tree-like intermediate representation is called a *parse tree* and is what GCC uses, in various forms, throughout the compilation process.

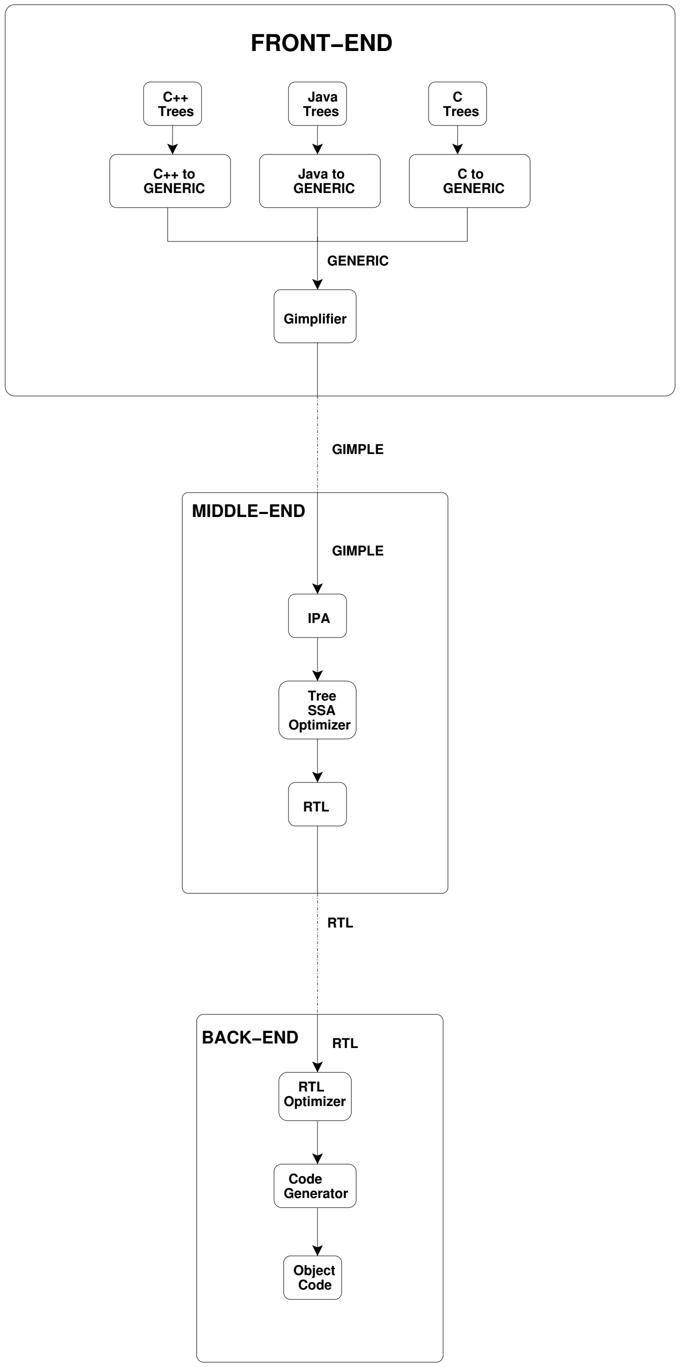


Figure 2: The GCC compilation process adapted from Red Hat Magazine. [35]

2.2 Middle-End

The middle-end in GCC was designed to perform virtually all architecture-independent optimizations. Before 2004, GCC was separated into two parts: the front-end and back-end. Whereas this worked in the past and is still how many other compilers operate today, GCC developers were running into problems. Following this two-phase design, optimizations such as loop unrolling and constant propagation were performed on a representation very close to machine code. Although not necessarily a problem for compilers supporting a small subset of languages or architectures, GCC developers found these optimizations were becoming quite difficult to maintain [25]. To simplify things, GCC developers separated optimizations from the rest of the code, giving them a separate compilation phase along with its own representation. In 2006, the GCC developers integrated support for inter-procedural optimization into the middle-end, further extending the capabilities of middle-end optimizations.

GIMPLE: GCC's middle-end optimizations begin with *Gimplification* of the initial parse-tree representation. Gimplification is the process of converting language-dependent parse trees into a simplified three address language-independent representation called GIMPLE. GIMPLE was named after, and is heavily influenced by, the McGill Compiler Architecture's language-independent abstract syntax tree representation, called SIMPLE [16]. Immediately after Gimplification, GCC constructs a *control-flow graph* (CFG) for each function consisting of a single entry and exit point, a set of nodes, and a set of edges connecting these nodes.

In addition, at this point GCC constructs a *call graph* which shows the function call structure. Each call graph node represents a function in the source base of the currently compiling code and has a list of callers and callees with a series of edges connecting the nodes. Together, these nodes and edges form a graph representing program function call semantics. An example call graph is shown in Figure 3. These higher-level structures allow for rapid control-flow and data-flow analyses. The simple nature of the individual instructions and the deterministic execution order inside a basic block also serve to make program analysis easier. Once all architecture-independent optimizations, such as loop unrolling, have been performed, we enter the back-end phase of compilation.

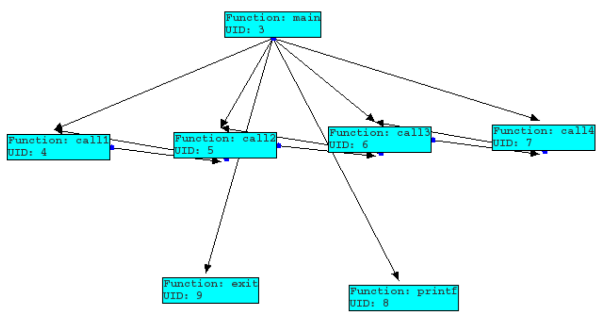


Figure 3: A subsection of a call graph rendered by GDE. Each node represents a particular function while edges represent function calls.

2.3 Back-End

The back-end is primarily responsible for generating the final assembly code for the program. In order to do this, GCC must allocate registers, perform final stack-frame layout, and schedule instructions for the CPU's pipeline. At this point, most optimizations have already been applied to the code and as a result, the only optimizations the back-end compilation phase need apply are architecture-specific optimizations, such as instruction pipelining. The back-end phase of compilation has been extensively developed over the years. As a result, modifications to this layer are now almost exclusively done for the purpose of porting or to improve GCC's exploitation of CPU resources.

3 Development Methodology

As mentioned in Section 1, code transformations allow developers to optimize and add functionality to code at compile time. Traditional development of code transformations, however, is a difficult process with several development obstacles to overcome.

The developer first needs to make sure the code transformation modifies the intermediate representation in such a way that file compilation is still possible. That is to say, the developer cannot break the compiler. Second, modifying the compiler source requires a full compiler rebuild, a process taking more than thirty minutes for GCC on an AMD64 X2 4400 dual-core [36]. Third, distribution of a completed transformation is very difficult requiring the user to manually modify compiler source files to apply the transformation. When applying more than one transformation, this is difficult at best due to the complexity of GCC source files. Fourth, transformation development requires the careful modification of a compiler's internal representation. This is highly nontrivial because that the internal representation becomes more and more low-level throughout compilation. Understanding the representation becomes harder as we get closer to assembly. Lastly, debugging a transformation is no easy task. Although a buggy high level application often has useful error messages, a buggy transformation usually has cryptic or short error messages which are of little help to an inexperienced transformation developer. The remainder of this section first describes GCC Plug-ins in Section 3.1, then describes a plug-in we have developed, DB-dump.

3.1 GCC Plug-ins

GCC plug-ins, which are scheduled to be included in mainline GCC version 4.5, give developers the ability to develop code transformations with modifications to the source base of GCC itself. Currently, developers need only to recompile GCC *once* to support the plug-in system and once plug-ins have been incorporated into mainstream GCC, no source modification will be necessary. GCC plug-ins are developed as separate files and then compiled into shared libraries which are loaded into GCC at run-time. This is done by the addition of function calls, which load arbitrary lists of plug-ins, at locations corresponding to individual phases of compilation. Figure 4 shows this process in more detail.

A user simply includes the flag *-ftree-plugin={Path to compiled plug-in.so file}* for each plug-in to be applied. The GCC plug-in system not only solves the problem of rebuilding GCC multiple times, but it also solves the transformation deployment problem; if a plug-in causes compilation to fail, simple remove it from the list.

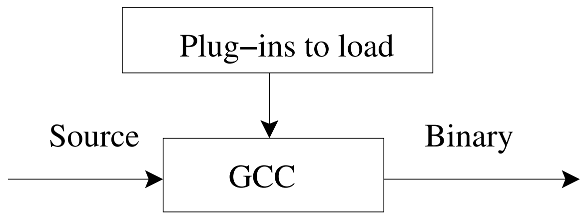


Figure 4: A figure showing the plug-in loading process.

While the plug-in system solves some of the problems associated with transformation development, GCC plug-ins do not make it any easier to understand a complex intermediate representation or to debug a broken transformation. As we will show, visualization of the intermediate representation ameliorates these problems. A compiler's intermediate representation is internal to the compiler, however, and in order to visualize the intermediate representation, we first must extract it.

3.2 DB-Dump Plug-in

DB-dump was a GCC plug-in developed to capture GCC's intermediate representation. The db-dump plug-in works by parsing a GCC definition file called *tree.def*, which contains a description of each element of GCC's GIMPLE intermediate representation. Using tree.def along with a custom definition file, we have designed, *parameter.def*; db-dump is able to recursively iterate through each element of the GIMPLE tree, storing node information at each step along the way. We chose PostgreSQL as the database system in order to keep with the open-source nature of GCC. We designed the schema to allow the efficient storage of GCC's complex intermediate representation along with useful source file information. We create tables for GCC internal items such as basic blocks, the call-graph, and the control-flow graph as well as for source file information such as functions, the actual source code of the file, and source-code statements. We also create tables for each type of GIMPLE tree node found in tree.def in order to keep table sizes manageable. Data replication was a major concern when we were designing db-dump as GIMPLE trees contain a lot of redundant type information. DB-dump handles data replication by only inserting new information into the database. When db-dump comes across data it has already seen, it creates pointer to the existing entry instead of creating a new entry.

3.2.1 Pointers

All pointers db-dump inserts into the database are hash values created through a two stage process. First we create 40-byte hash using a SHA-1 [9] hashing function with combination the file name, current function name, and the address of the current node being processed as input. Then, we attach a four-byte numeric description of the table we are going to insert into to the end of the hash and insert the 44-byte value into the database. The four-byte numeric description allows developers to quickly determine which table to search given a specific node while the hash value allows for quick lookups within that table. As Section 5 shows, our database system is able to handle complex source files efficiently.

4 Design

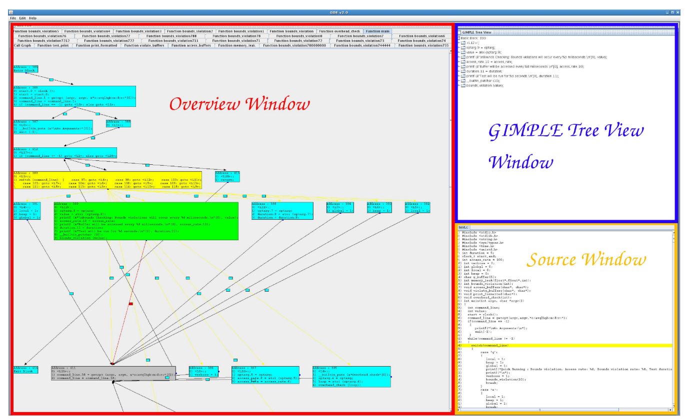


Figure 5: The GDE user interface

We have developed the Gimple Development Environment (GDE) using Java to visualize GCC's GIMPLE intermediate representation. We also have provided a graphical interface to the Gnu Debugger (GDB) which simplifies run-time plug-in debugging. GDE uses the Swing [11] library to render components, the AWT [27] library to draw decorations (e.g., lines connecting the elements of the CFG), the PostgreSQL JDBC driver [33] for database queries, and GDB for debugging. We chose Java as the development language for its cross-platform compatibility, which allows GDE to be used on most of GCC's host platforms. This allows us to concentrate on the development of the tool itself as opposed to platform support and library dependencies. As shown in Figure 5, GDE has three main areas: the overview window, the GIMPLE tree view window, and the source window, which we discuss in the following three sections. Finally, we discuss the graphical interface to GDB we have created in Section 4.4.

4.1 Overview Window

The overview window displays one of two main elements: a visual representation of the CFG of each function in the source file, or a visual representation of the call graph of the file. The call graph and each individual function are accessible via named tabs.

4.1.1 CFG:

As shown in Figure 6, the CFG is rendered as colored rectangles connected by arrows with flags associated with each edge. All elements of the CFG are movable and able to be minimized while the lines connecting each element of the CFG can be hidden. This allows the user to rearrange the graph at will to get a better view of a particular basic block of interest or to rearrange a loop into a form that corresponds better to high-level semantics. This also allow the user to hide uninteresting graph elements in order to better view an area of interest. Each colored rectangle corresponds to a specific basic block with a series of GIMPLE expressions to be executed in sequential order. Mousing over one of the flags associated with each edge causes the flag to expand, displaying the GCC edge flags associated with that particular edge.

Edges and edge flags: As discussed in Section 2, basic blocks in the CFG are connected by directed edges which specify the control flow through the graph. Most edges, with the exception of the edge from the last basic block to the exit block, have a set of one or more flags associated with it. These flags specify when a particular node is taken. For example, the `EDGE_FALLTHROUGH` flag specifies that this edge is taken at all times, whereas the `EDGE_TRUE_VALUE` flag specifies the edge is only taken when the conditional in the previous basic block evaluates to true.

Clicking a CFG node here has several effects. First, GDE colors the selected block green, while coloring its predecessors yellow, and its successors gray. GDE also

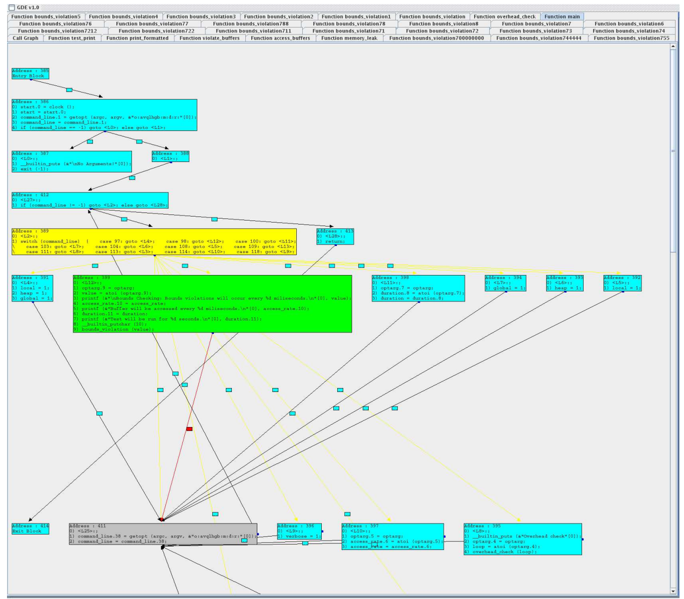


Figure 6: The CFG rendered by GDE

highlights the paths to each successor in red. This allows the user to quickly determine which blocks could follow the execution of this block and also which blocks could have preceded it's execution, which allows for easy flow analysis. Second, GDE displays a visual tree representation of the selected basic block's GIMPLE nodes in the GIMPLE tree view area. Finally, GDE highlights the lines of source code corresponding to the selected basic block, its successors, and its predecessors in the source area.

4.1.2 Call Graph

As shown in Figure 7, the call graph is comprised of colored rectangles connected by arrows. Each colored rectangle here represents a node in the call graph for a particular file and each edge represents a function call from one node to another. Nodes in the call graph simply contain a unique identifier assigned to that node along with the name of the function that the node represents. All call graph elements are moveable, able to be minimized, and the edges connecting each node can be hidden. We have implemented this functionality for the same reasons discussed in the CFG segment above. Clicking a node of the call graph causes that node to be highlighted in green, any node called by that node to be highlighted in gray, and any nodes calling the selected node to be highlighted in yellow. Paths to each node called by the selected node are also highlighted in red by GDE, similar to the highlighting scheme of the CFG described above.

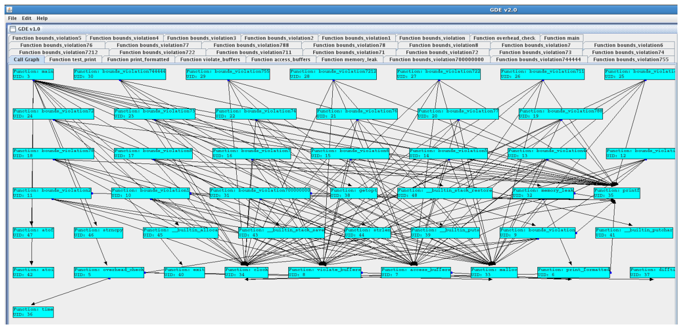


Figure 7: The call graph rendered by GDE

4.2 GIMPLE Tree View

When a control-flow graph is being displayed in the overview window, clicking a basic block displays its corresponding GIMPLE representation in the GIMPLE tree view. The root node of each tree is a statement from the corresponding basic block rendered in a C-like syntax. The tree generated is a visual representation of the attributes and operands for the selected GIMPLE node, as previously discussed in Section 2. Non-leaf nodes are GIMPLE attributes or operands that have at least one pointer to another node, whereas leaf nodes represent nodes that have no pointers to other nodes. The tree view is useful as it visualizes the ordering of operands in each node and also lets the developer know what attributes apply to a particular node. This is invaluable when using macros such as `TREE_OPERAND`, which programmatically dissect tree nodes, inside GCC transformations.

Clicking a node in the GIMPLE tree view expands that node, showing its children. Each non-leaf child node can then be expanded, in the same manner, until the desired information is found. Initially, clicking a basic block caused the GIMPLE trees to be created for all statements in the basic block. This meant recursively visiting each node in each tree in the selected basic block, creating the visual objects at each step along the way. Although this worked for most basic blocks, as Figure 8 shows, larger basic blocks were simply too large to be rendered in their entirety.

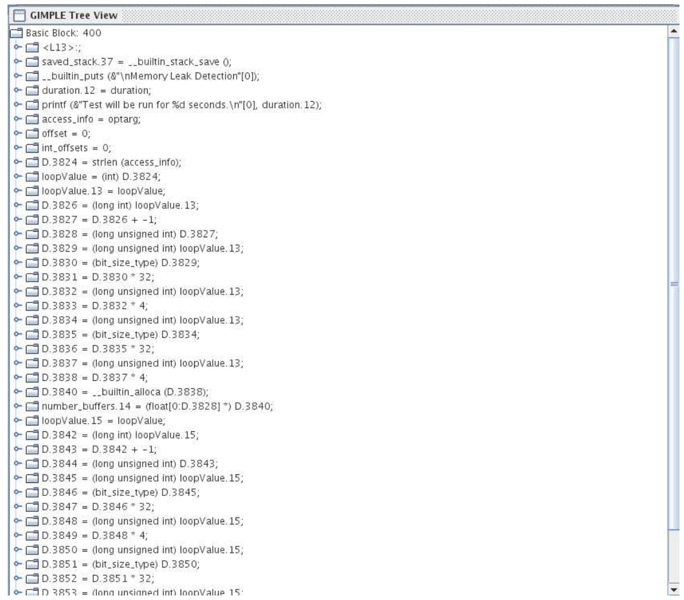


Figure 8: An example basic block with more statements than usual shown in the GIMPLE Tree view of GDE.

Furthermore, due to the size of the GIMPLE, medium to large sized basic blocks were taking a noticeable amount of time to render.

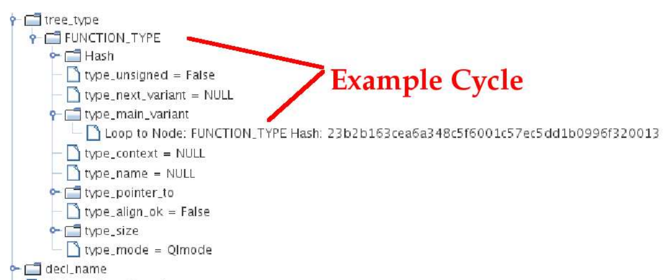


Figure 9: An example cyclic GIMPLE access, the cycle is detected and reported by GDE.

To address this, we implemented dynamic GIMPLE tree construction. Now, clicking a basic block causes only the queries necessary to create the top level nodes to be executed. We then used the results of those queries to create visual representations of each top-level node. We create visual representations of child node in the same way as the user expands each parent node. This allowed us to remove the tree depth limit but forced us to deal with another problem that had previously been handled by the depth limit. Although GIMPLE is best visually represented as a tree structure, GIMPLE nodes can occasionally form cycles when a child node points back to a parent node, as shown in Figure 9.

Although these cycles do not occur often in each particular GIMPLE tree, they exist in every GIMPLE tree. Without the depth limit, a user could potentially enter one of these loops and expand the tree until GDE runs out of memory. We have addressed this issue by adding loop detection as we create the GIMPLE tree; instead of blindly displaying tree nodes, we instead display nodes only if they have not previously been rendered. When a node is previously displayed, we inform the user that the node is a back reference using a placeholder node which contains the hash of the back reference.

4.3 Source Window

The original source file, corresponding to the intermediate representation currently being examined, is displayed in the source window with line numbers for quick reference. Although the user cannot explicitly interact with this area, clicking a basic block in the CFG of a function highlights the line(s) of code corresponding to that basic block in green, the line(s) corresponding to its successors in gray, and the line(s) corresponding to its predecessors in yellow. This allows the user to easily identify which lines of code in the source were compiled to produce a particular basic block, explicitly displaying the source-to-intermediate representation mapping.

4.4 GDB Console

GDE has the ability to debug a plug-in as it runs using our GDB console. As a running plug-in is loaded into GCC, debugging a plug-in requires the user to debug GCC itself. Although most binaries can be debugged by attaching a debugger to the running binary, debugging GCC is not as straightforward. The command *gcc* is not the actual GCC compiler, but instead the compiler driver which determines the type of file being compiled, sets several arguments normally transparent to the user, and finally calls the appropriate compiler to compile the source file. We show this process in Figure 10. To debug GCC, the user must attach the debugger to the correct binary while also setting the same arguments that the GCC script would set. We have automated this process by simply opening the GDB console from within GDE.

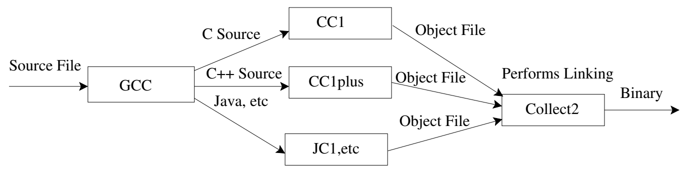


Figure 10: The GCC calling process. Actual file compilation and linking are done by files called by the *gcc* compiler driver.

As Figure 11 shows, the GDB console has five areas of interest: (1) the CFG area, (2) the GIMPLE tree window, (3) the backtrace window, (4) the GDB output area, and (5) the Input area. The GDB output area displays all output from GDB as received along with occasional GDE output used mainly for GDE debugging purposes. The input area is where the user interacts with the underlying GDB debugger. Users are given a dropdown box with GDE commands, a text input area, and several buttons corresponding to common commands.

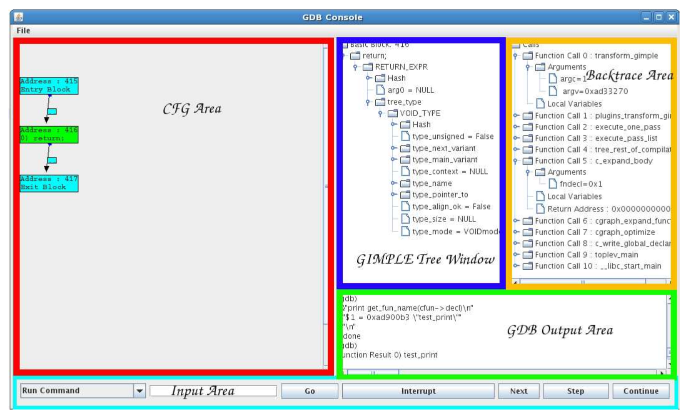


Figure 11: The GDB Console of GDE.

When a user selects the dump function option while GDB has stopped inside a function, GDE creates a visual representation of that function's CFG in the CFG view. Clicking CFG nodes in the CFG view has the same result as clicking a CFG node in the overview window of GDE as described above: the GIMPLE tree is displayed in the GIMPLE tree window of the GDB console. The generated GIMPLE tree, however, is a snapshot of the current state of the intermediate representation. This allows developers to see any changes that happen to the GIMPLE tree as they happen, giving insight into where a plug-in may be operating incorrectly. Lastly, selecting the backtrace option displays the results of running backtrace command in the backtrace window in a more readable form.

5 Intermediate Dump Analysis

This Section discusses the results of several intermediate dumps using the db-dump GCC plug-in we described in Section 3. As we show, being able to dump the intermediate representation of compiling source code using our db-dump plug-in allows for static analysis to be performed on that data at a later time. To begin, we discuss the files examined, including a brief explanation of each file. We then discuss our dump statistics and conclude with a discussion of some types of analysis possible on our db-dump output.

5.1 Files Examined

The first file we dumped was a test file created mainly for the development and debugging of plug-ins, `test.c.reference`. This file is a simple file, written in C, which simply computes the factorial of a number in a tabular manner. Next, we dumped a second internal file named `test.c.benchmark`, which is based off a file used to test for bounds violations. It does this by accessing in bounds and out of bounds areas in the stack, heap, and global areas at a user specified rate. This file has been modified slightly to increase the overall size of the file by the addition of function copies, which was done to test the visualization capabilities of GDE with respect to a larger single input file. We then dumped two real world projects: Lighthttpd [19] and the Linux kernal [39]. Lighthttpd is a light weight web server written in C. The first Linux kernel configuration we have dumped was created using the make `allnoconfig` option, which turns off as many features as possible. We then turned on only the Ext2 filesystem and lock debugging utilities for our next configuration.

5.2 Dump Sizes

Table 5.1 shows our dump sizes. We give numbers for: (1) the overall size of the database, (2) the number of functions compiled and dumped, (3) the number of three address statements compiled and dumped, (4) the number of basic blocks dumped, and (5) the total number of source lines (including non compiled items such as comments).

Name	Size(kB)	Number Functions	Number Statements	Basic Blocks	Source Lines
kernel-ext3-lock	817000	15826	286512	75660	271120
kernel-allnoconfig	678000	13435	241778	65860	241014
lighttpd	87000	2310	45516	11037	36321
test.c.benchmark	2688	46	2097	414	1139
test.c.reference	5	43	16	39	

Table 1: Showing DB-dump key statistics

Figure 12 shows the relation of size vs. number of statements dumped for all files. This is a good metric of how our database system scales with project size. As the lines of C code in a project increases, the number of statements corresponding to those lines of C code generally increases as well. As each statement is the starting point of a GIMPLE tree, the more statements you have, the more GIMPLE there will be corresponding to those statements. The large ratio of size vs.statements for `test.c.reference` shown in Figure 12 is due to the small size of `test.c.reference` itself. When looking at small files, the database declarations alone cause the size vs.statements ratio to be very large. However, it should be noted that in this case, even though the ratio itself is large, the actual size of the database is only 664kB. Figure 13 is a better metric of the scalability of our system. As this figure shows our database size scales linearly with the number of statements.

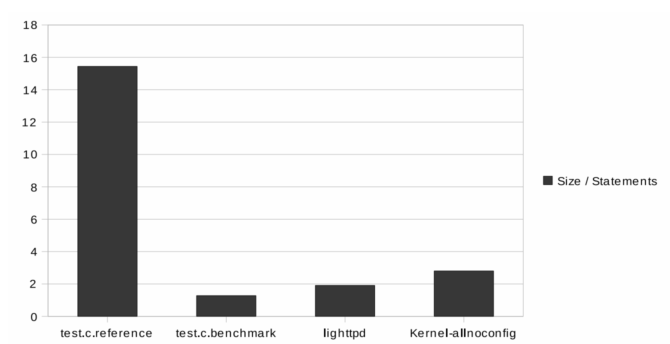


Figure 12: Database size vs.number of statements. Although the ratio is high for `test.c.reference`, the overall size is 664kB.

5.3 Potential Uses

In this section we give potential analysis that can be done on the intermediate dump of a program. We start with a discussion of complex networks, what they are, and why complex network analysis is useful. We then discuss how analysis can be done on a software system to determine if usage conventions are being followed properly. Finally, we discuss model checking and how tools could be used with db-dump to verify system properties.

Complex networks: Complex networks are defined as network exhibiting non-trivial topological properties. The process of determining if a network is a complex network involves examining the structure of the network to determine if the network has these properties. Many real-world systems have been shown to exhibit complex network properties such as predator-prey interactions between species in a freshwater lake, neural networks, and networks of citations between papers [29]. Developers can perform complex network analysis on the control flow graphs and call graphs extracted by db-dump. Once a network can be shown to be a complex network, certain assumptions can be made which may have a large impact on both system security and recoverability [22]. Developers can do this analysis off-line with the results then used to target specific areas of a large software system for improvement.

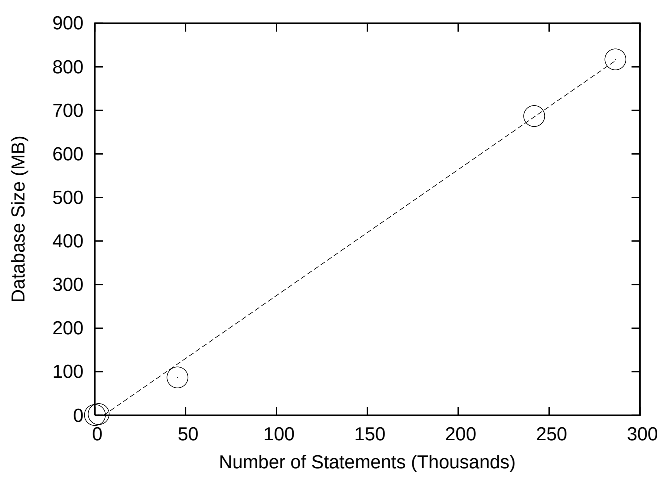


Figure 13: Database size vs.number of statements for each test file.

Code Convention: In large software systems, item usage is often done through convention and is not strictly enforced. For example, when accessing certain structs within the Linux kernel, certain locks *should* be taken. This locking policy is not strictly enforced in all areas and as a result, some structs are accessed without the appropriate lock being taken first. While this usually has no affect on the overall operation of the system, occasionally it can lead to race conditions. Our schema was designed in such a way that it is possible to write relatively simple queries to target specific node types. This allows developers to perform off-line analysis of a system to look for things like the usage patterns specified above to determine if access conventions are being followed correctly.

Model Checking: Symbolic model checking allows the verification of many non-trivial properties of large software systems. Tools exist, such as NuSMV 2 [8], to allow developers to verify questions about system security without having to learn or implement complex model checking methods. The information stored in our database represents the internals of an entire code base. Developers can format this data appropriately and pass it to model checking tools to verify the correctness of portions of the system.

6 Use Cases

In this section we give example uses for GDE using the db-dump plug-in to extract GCC's intermediate representation. We discuss several plug-ins that we have developed in order to illustrate the benefits GDE brings to plug-in development. The three plug-ins we use as examples in this section are a call-tracing plug-in, the verbose-dump plug-in previously discussed in Section 3, and a bounds-checking plug-in. Our first two use cases discuss issues faced while developing a call-tracing plug-in. We then discuss an issue faced while expanding the predecessor to db-dump, a plug-in named verbose-dump. Verbose-dump is very similar to db-dump with the differences being verbose-dump outputs GIMPLE to stdout and does not handle redundant data well. Next, we discuss issues faced while developing a bounds-checking plug-in and we finally conclude by discussing a potential use case for our GDB console.

6.1 Dissecting GIMPLE Trees

Our call-trace plug-in is written in C and adds tracing to a program without modifying the program source code. It does this by finding specific GIMPLE nodes we are interested in logging, then extracting the information we want to log from those nodes. As we show, GDE helped the development of this plug-in.

When writing the call-trace plug-in, to target specific GIMPLE nodes it was necessary to find and replicate intermediate representation patterns corresponding to those nodes. For example, we wanted to add functionality to the call-trace plug-in to detect and log conditional statements. We were interested in reporting that executing code had reached a conditional and what the conditional evaluated to: true or false. To do this, we needed to figure out how conditionals are expressed in GIMPLE in order to target conditional nodes with our plug-in. Checking *tree.def* gave us some information about conditionals, but the information contained was vague, stating operand one was the then-value while operand two was the else-value. However it did *not* tell us what those operands were. They could have been one of many nodes, each requiring a different approach for field access. Using the steps we describe below, we show how the GIMPLE tree view of GDE made the task of finding what the operands were easier than the traditional approach.

1. We began by writing a simple test case, using C, containing several conditional statements. We then compiled the test case using GCC along with the db-dump plug-in to dump the GIMPLE intermediate representation to our database. Once the dump was complete, we looked at the intermediate representation stored in the database using GDE.

2. When inspecting the GIMPLE representation of the code, our first task was to locate a conditional statement in the CFG in the overview window. Once we found a block with a conditional statement, we clicked it to display the GIMPLE tree in the GIMPLE tree view window. As Figure 14 shows, we were quickly able to see that the type of node corresponding to a conditional expression was a `COND_EXPR` node. Further, we were able to see that the first operand of a `COND_EXPR` (or conditional expression) node was the actual conditional test itself (in this case, an `EQ_EXPR` or equality test), followed by the left and right branches of the conditional. It was here that we were able to see that the operands were `GOTO_EXPR`s. Using this information, we were able to design our call-trace plug-in to add logging statements in the correct basic blocks to indicate if the left or right branch was taken.

As this example demonstrates, finding and reproducing simple GIMPLE code patterns is non-trivial. In this case we were looking for all conditionals. It is clear that if we were interested in a subset of conditionals, containing a specific variable for example, then our code pattern would become more complex and harder to find without the aid of a visualization tool such as GDE. We show a more complex example in Section 6.2.

6.2 Dissecting Complex Expressions

Generating complex GIMPLE expressions programmatically can be difficult for even experienced programmers due to GIMPLE's low-level nature. It can be unclear exactly how certain items are represented in GIMPLE. For example, while adding function call logging to the plug-in, we were interested in printing the fields in pointers to structs being used as function parameters. To do this, we first needed to reliably identify function calls with at least one pointer to a struct as a parameter. We used GDE to accomplish this in the following way.

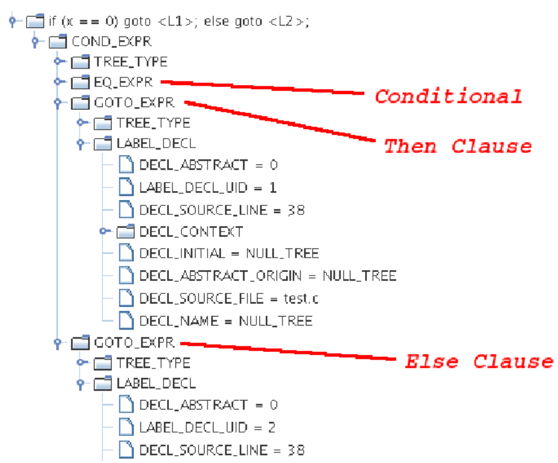


Figure 14: Using GDE to get information about a COND_EXPR.

1. As before, we wrote a small test case in C containing the fragment of code we wanted to generate: in this case a function call with the address of a `struct` as a parameter. We then compiled our new test case with the db-dump plug-in enabled, and inspected the output in GDE.

2. As shown in Figure 15, we were able to see exactly how this particular statement was represented in GIMPLE by GCC. In this case, the function call was a `CALL_EXPR` node with several subtrees, the last of which was an `ADDR_EXPR`. This indicated that the node is a reference to the address of an object, which is what we were looking for.

3. As we dug deeper, we discovered the `ADDR_EXPR` node pointed to a `VAR_DECL` node, which indicates a variable. Finally, examining the `TREE_TYPE` attribute of the variable told us that the variable is of type `RECORD_TYPE`, showing that GCC represents a struct as a `RECORD_TYPE` node. This information about how GCC represents these kinds of function calls allowed us to write code that reliably identified them.

Generating complex expressions can also be done by hand after sifting through GCC source files. This would be a long and tedious task, due to the different types of attributes and operands that each node contains. Any mistakes made during translation would likely be difficult to track down later due to the cryptic nature of compiler errors.

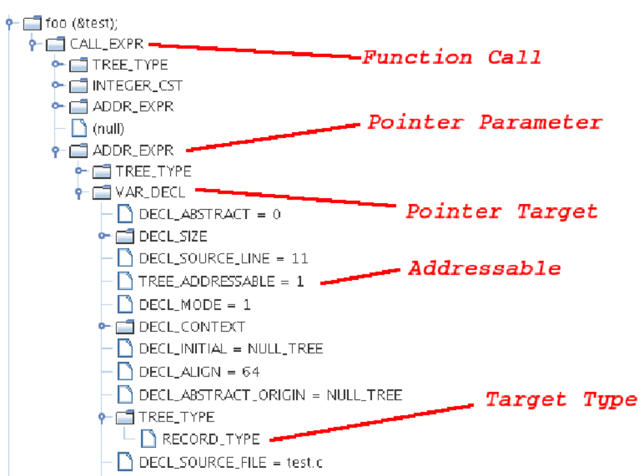


Figure 15: Using GDE to see how a particular statement is gimplified.

6.3 API Usage

The GCC API relies on specific macros, functions, and objects to access nodes and node data. Whereas some items like `TREE_TYPE` can be used very generally, others like `TREE_CHAIN` are specific to a particular kind of node, causing an error otherwise. GCC is complex and the GCC internals documentation is incomplete and frequently out of date with respect to the most recent release. As a result, a person unfamiliar with GIMPLE can spend hours trying to figure out how to access a particular field or child-node. GDE speeds up this process significantly by providing insight into what might be needed for a particular node access.

When we were expanding the verbose-dump plug-in to print the C parse trees for functions we were unsure how to iterate through the list of statements in a nested block. When we inspected the node corresponding to the nested block in the GIMPLE tree view, we found that it had a `STATEMENT_LIST` operand, as shown in Figure 16. Before we did this, it was not clear to us exactly how this list was stored; it could have been a `TREE_CHAIN`, which requires the use of a macro to access each element. As it was a `STATEMENT_LIST`, we knew that we had to use the *tree_stmt_iterator* object to access each element of the list. Using GDE in this situation helped us to figure out exactly how to access the information contained within that node.

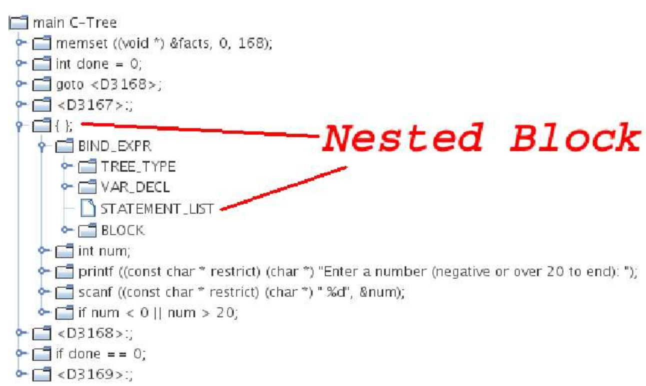


Figure 16: Using GDE to help determine which macro to use.

6.4 Debugging Bad Code

Our bounds-checking plug-in adds run-time bounds-checking to a source file by looking at pointer dereferences and checking if those references point to a valid memory area. While developing this plug-in, we ran into several issues that GDE was able to help with.

Even when the programmer understands what needs to be done, GIMPLE programming is error-prone. The difficulty is compounded by the fact that errors are typically caught much later in the compilation process and generate cryptic error messages. For example, we have found that most malformed GIMPLE code simply causes a segmentation fault in GCC which gives the error message *internal compiler error*. Debugging is made easier when the GIMPLE information is visualized with GDE.

For example, our bounds-checking plug-in declares an array variable containing all of the addresses of stack areas declared by each function for use by the bounds-checking runtime. Although everything seemed to be written correctly, using the plug-in was causing an error to be generated rather late during compilation. Looking at the code in GDE, we found through trial and error that if we attempted to record the address of variables that did not have the `TREE_ADDRESSABLE` flag set, the compiler would crash. We found out that the flag indicates that an item has a valid address. It was only through the use of GDE that we were able to determine that the flag was the problem. To fix things, we simply did not record the address of variables with the flag unset.

6.5 CFG Inspection

In this section, we discuss two use cases concerning the CFG, basic block inspection and edge inspection. Both use cases involve the bounds-checking plug-in described earlier.

Basic Block Inspection: Our bounds-checking plug-in can dynamically switch bounds-checking on and off. To do this, the plug-in replicates the entire CFG for each function while also inserting an additional basic block to each CFG that chooses to execute either our instrumented code path or the original uninstrumented path. While developing this bounds-checking plug-in, however, we found the transformed code was not executing properly.

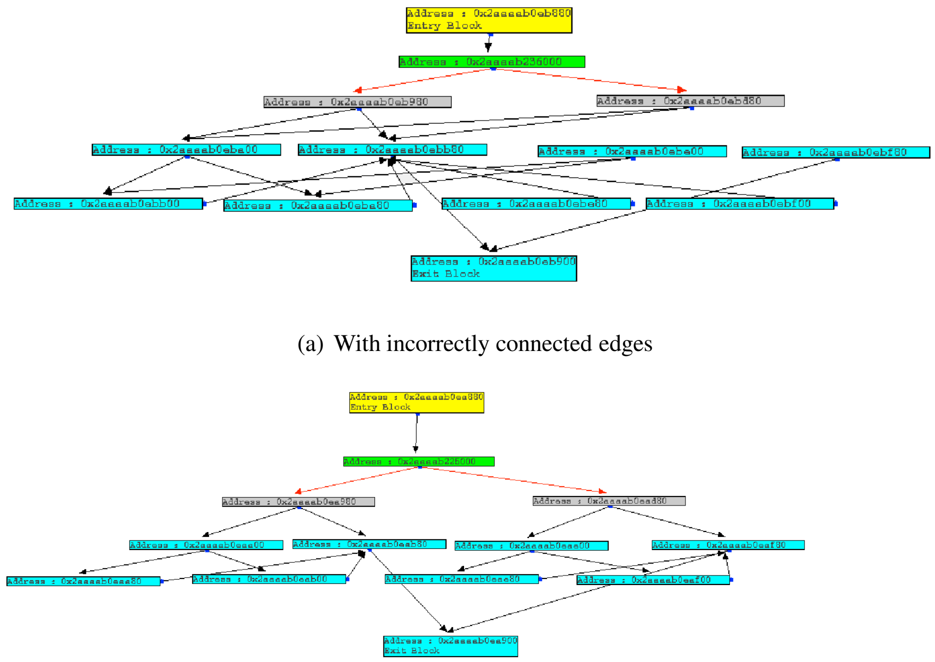
Figure 17 we shows both the CFG generated by the buggy version of the CFG duplicator as well as the correct version produced after the bug was fixed. We have minimized all the CFG nodes to show only the structure of the CFG.

We were trying to generate a duplicate CFG with identical left-hand and right-hand sides except for two shared initial and ending nodes (the top and bottom nodes) as well as a node to decide which path to take. As Figure 17 shows, all basic blocks were being replicated correctly. As this use case demonstrates, GDE can be useful in not only figuring out what is the problem, but also what isn't the problem.

Edge Inspection: As we have shown above, using GDE we found that although the nodes of the graph were being replicated properly, the problem was that the edges connecting the nodes were not. All outgoing edges were incorrectly connected to nodes in the left-hand copy. The alternative to using GDE would have been a very difficult task requiring parsing of the intermediate representation to create the CFG by hand or designing elaborate test cases to see in which cases code executed properly. However, the overview provided by GDE immediately illustrated the problem, and we were able to correct the graph which fixed the problem.

6.6 Run-time GIMPLE Inspection

This section presents a hypothetical use case for our GDB Console. While developing plug-ins, it is often necessary to debug GCC itself. As we have stated in Section 4, that process requires more effort than debugging a typical program, and even once that is done, extracting run-time GIMPLE information is a non-trivial task. Through the use of the GDB console, developers have the ability to look at GIMPLE with the click of a mouse. For example, if a developer wanted to create and insert a new `COND_EXPR` node into a particular basic block, the developer would have to construct the `COND_EXPR` node first, along with the operands. If the developer performed this construction incorrectly, perhaps by specifying the condition node incorrectly, the result would most likely be an internal compiler error



(a) With incorrectly connected edges

(b) With properly connected edges

Figure 17: Invalid and valid versions of a duplicated control-flow graph.

when the developer attempted to compile the program. If this was occurring in only one location, it might be possible to track the problem down quickly. However, most transformations work by modifying or adding several nodes, not just one. If nodes are being correctly being created in most places, but incorrectly in others, perhaps due to a cascading problem, then tracking down the problem becomes much more difficult. Using the GDB console, it would be possible to look at the GIMPLE at each step of the transformation. The developer would be able to see a snapshot of each GIMPLE tree as it currently exists during compilation, which may provide insight into the problem.

7 Related Work

Graphical development tools and debuggers simplify many elements of application development by allowing the developer to debug or develop an application visually. In this section we discuss tools in three categories. In Section 7.1 we discuss graphical tools for program development. In Section 7.2 we describe compiler visualization tools. Lastly in Section 7.3 we briefly discuss the C intermediate language (CIL), a C-like language that allows developers to develop source to source transformations, and its uses compared to traditional transformation development.

7.1 Graphical Development

Graphical Debuggers: Stand-alone graphical debuggers, such as GNU DDD [15] or GDBX [4], are designed to cut development time by allowing the developer to view source code along with some visual representation of the run-time data of that code. Often, these tools are designed to provide visual information to the user by visualizing the output of a command-line debugger such as GDB [12] or dbx [38]. This use is common enough that some debuggers have output modes used when the debugger is part of a larger system. GDB, for example, supports a special mode called *machine interface* mode, which automatically formats GDB output to be easily parsed by a front-end. However, not all tools operate in the manner and instead choose to directly modify an executing binary. Development environments, such as Eclipse [40], provide debugging information to the developer along with a set of other development tools, such as a source-code editor. Whereas it may be simpler to parse GDB output, binary modification allows the developer to do things like *hot swapping* executing code; modifying executing code without a full rebuild of the binary. Over the years, other debuggers have also implemented visualizations and are similar to the systems described above. The SoftBench [17] and CodeCenter [7] debuggers, for example, support simple data structure visualizations in the form of box-and-arrow diagrams. Integrated and stand-alone graphical debuggers such as these are useful as their visualizations make it easier to pass input to and to view output from the debugger. These tools do this by providing an interactive debugging interface to the user, allowing the user to set breakpoints, set watches, and view run-time data visualizations through mouse clicks. Although ease of input through mouse use may not be all that useful to a highly experienced command-line debugger user, it may be highly beneficial to a less experienced debugger user. The run-time data visualizations these tools provide may give insight into problematic areas of code; useful to both experienced and inexperienced users. Although plug-ins could be debugged or visualized with these tools, they are very general purpose, designed to work on a variety of programs. GDE, on the other hand, has been designed specifically for use with plug-ins.

UML Tools: UML tools, like Rational Rose [34] and Visio [26], allow developers to specify items such as class relations, local variables, or function prototypes for a particular application in a visual manner. This allows the developer to see a high-level representation of the application which often gives insight into any shortcomings in its design. When the developer is satisfied with the application layout, a simple button click creates a skeleton of the program.

Graphing Tool-kits: Graphing tool-kits allow the visualization of data. Tools like aiSee [1] work by reading input specified in a custom graph description language, then creating and visualizing a graph based on the input specified. Some tools can be used by other programs to perform visualization. Doxygen [10], for example, creates documentation for a source package by scanning source code and parsing directives found in the source code of a package, similar to using javadoc [37] on a Java file. When configuring Doxygen, users are given the option to create a visual representation of the scanned sources if they have GraphViz [3] installed. Other tools, such as Program Explorer [23] and Module Views [42] also exist and provide data visualizations for object oriented programs. These visualizations include call visualizations, object creation visualizations, execution visualizations. Lastly, projects such as the *Jinsight* project are interested in examining the dynamic behavior of Java programs [18]. The *Jinsight* project have developed Java specific visualizations, such as object visualizations to find wasted memory [32] and a method call visualization tool [31], to examine this behavior. These visualization tools like these are useful because they give the developer a high level, concrete view of the interactions of an application. This in turn may give the developer insight into problematic areas of the application's design or may be able to give insight into debugging an application. Although these tools are useful, they are general purpose and require either the learning of a graphing language to describe their graph or the insertion of directives throughout program code for data visualization. The run-time information these tools provide is not suited to plug-in development due to its high-level nature.

7.2 Compiler Visualization

Whereas graphical development tools have been shown to vastly improve application development by displaying complex information in an easy to understand form, little has been done to visualize complex compile-time data.

The Interactive Compiler: The Interactive Compiler [41] was one of the first attempts at visualizing compiler information. It is a custom compiler written in Smalltalk-80 which compiles a simple language consisting of assignments and conditionals. After the initial compilation, the interactive compiler generates an intermediate representation (IR) which is then displayed to the user, in text-based form, and can then be edited as needed. Although the interactive compiler laid the groundwork for much of what we have done, there are two issues which make it unsuitable for use as a transformation-development tool. First, due to technology limitations at the time, the IR information generated by the Interactive Compiler is displayed in a textual form. As we have shown in Figure 1, this is problematic when dealing with modern programs, as each line of source code produces many lines of IR output. Second, the compiler itself is only able to compile a simple language on a limited number of architectures, whereas transformation developers want to target compilers that can compile several complex languages on many different architectures.

***xvpodb and VISTA*:** *xvpodb* is a visualization tool developed to visualize the optimizations performed by the Very Portable Optimizer (*vpo*) [5]. *Vpo* is an optimizer designed to perform many low-level RTL optimizations [43]. These types of optimizations are things such as instruction selection, instruction scheduling, and dead variable elimination. When a file is compiled with *vpo* enabled, various *vpo* messages are intercepted by the *xvpodb* tool and saved in a file for later viewing. The user can then step forward and backward through the *vpo* optimization process, choosing to examine various pieces of information at will. This allows the user to see things like which transformations affect a specific instruction.

VISTA is a tool based off based *vpo* and designed to allow performance tuning of applications [20]. *VISTA* allows the user to step through transformation as *xvpodb* while also providing useful features such as source correlation via line highlighting. Lastly, *VISTA* is able to rate the effectiveness of optimizations and select the set of optimizations providing the best performance gain.

Although *xvpodb* and *VISTA* are useful in their own right, especially as teaching aids, they have one major drawback: they can only visualize the transformations performed by the *vpo* optimizer. This means the user is only able to look at RTL-level transformations, not transformations performed on high-level IRs. Whereas RTL-level transformations are very powerful, certain transformations, like function call logging, are better suited to high-level IRs. These tools, by design, are unable to visualize or modify non-RTL-level transformations.

7.3 C Intermediate Language

The C intermediate language (CIL) is a source-to-source transformation of C programs [28]. CIL users first write a transformation using CIL which is then applied to a user-specified source file. This combination creates a new C source file which is then passed to GCC to compile as usual. The main advantage of using CIL is that it allows developers to specify transformations using a simplified version of C; this means that developers need not learn a complex IR to add new functionality to existing code. Although CIL is a useful and powerful language, it has an inherent problem which limits its usefulness. CIL transformations by design only support source-to-source transformations of C programs whereas other transformations, such as GIMPLE transformations, are language independent. Using languages like CIL to perform transformations can quickly become cumbersome, requiring developers to learn a new language for each language they want their transformation to support.

8 Conclusions

Code transformations have traditionally been difficult to develop, requiring developers to directly modify the source files of a compiler, a highly non-trivial task. Deployment of a completed transformation is hard, necessitating a line-by-line addition of the transformation code to the existing source to ensure compatibility with other transformations existing on that particular system. GCC plug-ins have solved the problem of transformation deployment, but have not addressed the issue of transformation development.

Visual development is the solution to this problem. It has had great success in the past with debuggers, development environments, and modeling tools. We have presented the GIMPLE Development Environment, a useful tool to reduce the time taken to design, development, and debug GCC plug-ins and optimizations. We have also presented a GCC plug-in which stores the internal representation of a program in a database; a useful tool in its own right as we have shown in Section 5. The graphical control flow graph GDE creates for each function allows the developer to track the flow of information through a particular program from beginning to end much more effectively than the traditional method, looking at a text-based control flow graph information. Section 6 shows how this visual representation of the CFG aids in the debugging of plug-ins modifying the structure of all or part of an existing control flow graph. The call graph visualization capabilities of GDE allow developers to quickly determine predecessors and successors to a given function, and help with program data flow understanding. GDE's GIMPLE tree view allows developers to visualize the various GIMPLE trees for each statement in each basic block. This not only gives insight into which macros to call on a given node, but also allows for quick inspection of a transformation, allowing the developer to quickly determine if GIMPLE nodes are being modified properly. Lastly, our GDB console allows developers to examine the GIMPLE and control flow graph of a function as it is compiling, providing more useful information to developers as opposed to cryptic errors as discussed in Section 6.

We have found that although transformation development is inherently difficult, the use of these visual aids alleviates many of the difficulties associated with using the GCC Internals API and greatly lessens development time.

9 Future Work

In this Section we will discuss future research areas for GDE.

9.1 Zooming

Although having each component of GDE rendered in its own view is useful and functional, the call graph, control flow graph, basic blocks, and GIMPLE are all inherently related. We plan to modify GDE to use a zooming-based view. Initially, the user would be presented with the call graph, which the user could use to identify functions of interest. Zooming in on these functions would then give the user the control flow graph for that particular function, showing all basic blocks. If a basic block were particularly interesting, the user could then zoom in to view the statements and the GIMPLE for that block. This would expand and improve GDE usefulness with larger files.

9.2 Libraries

While GDE currently does not have the ability to examine pre-compiled libraries, it would be possible to extend GDE to handle them. This task would require either storing the compile-time information of all shared libraries in a database when each library is compiled, or dynamically re-compiling a library to obtain the compile-time information for that library. Dynamically re-compiling libraries would be a difficult task as the source might not be available for a given library.

9.3 lxr++

The Linux Cross Reference(LXR) allows developers the ability to quickly index and browse source repositories, with Linux kernal source browsing being one of the more useful features of the system. While this tool is very useful, it only gives developers a top level view of their code. As db-dump stores the GIMPLE information of a source base, extending LXR to also provide compile-time information for each statement would be useful to developers and straight forward to implement.

9.4 Online Functionality

Making GDE a web application is a practical and attainable goal. Although GDE is written in Java and requires little work to port from system to system, db-dump is a C++ GCC plug-in, requiring a specific configuration for each system it is to run on. Furthermore, the user needs to have Postgresql running on the system db-dump is running on. By putting GDE online, developers would only need to connect to a server, upload their code, and view it with GDE. This goal is particularly interesting as GDE is written in Java, converting GDE to an applet will not require a rewrite of the entire system.

9.5 RTL

We plan to further expand the amount of compile-time information displayed to the developer by visualizing the RTL of each function. RTL is used extensively by developers porting GCC between architectures and for developers working on improvements to GCC's code generator. Visualizing this level may greatly reduce the complexity of writing RTL code.

References

[1] AbsInt. aisee, 2008. `http://www.aisee.com`.

[2] R. Agrawal, L. G. Demichiel, and B. G. Lindsay. Static type checking of multi-methods. In *ACM SIGPLAN Notices*, 1991.

[3] AT&T Research Labs. Graphviz, 2009. `http://www.graphviz.org`.

[4] D. B. Baskerville. Graphic presentation of data structures in the DBX debugger. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1985.

[5] M. Boyd and D. Whalley. Graphical Visualization of Compiler Optimizations. *Journal of Programming Languages*, 3(2):69–94, 1995.

[6] S. Callanan, D. J. Dean, and E. Zadok. Extending GCC with modular GIMPLE optimizations. In *Proceedings of the 2007 GCC Developers' Summit*, Ottawa, Canada, July 2007.

[7] CenterLine Software, Inc. *CodeCenter Tutorial*, 1995. `http://products.ics.com/products/codecenter/codecenter-4.1.1-tutorial.pdf`.

[8] A. Cimatti, E. Clarke, E. Giunchiglia, F. Guinchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. *Computer Aided Verification*, pages 241–268, 2002.

[9] P. A. DesAutels. SHA1: Secure Hash Algorithm. `www.w3.org/PICS/DSig/SHA1_1_0.html`, 1997.

[10] Dimitri van Heesch. Doxygen, 2008. `www.doxygen.org/`.

[11] A. Fowler. A Swing Architecture Overview. Technical report, Sun Microsystems, 2007. `http://java.sun.com/products/jfc/tsc/articles/architecture/`.

[12] T. Free Software Foundation, Inc. GDB: The GNU Project Debugger. `www.gnu.org/software/gdb/gdb.html`, January 2006.

[13] T. GCC Team. The gnu compiler collection. `http://gcc.gnu.org`.

[14] T. GCC team. *GCC online documentation*, December 2005. `http://gcc.gnu.org/onlinedocs/`.

[15] GNU Project. The Data Display Debugger. `http://www.gnu.org/software/ddd`.

[16] L. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Lecture Notes In Computer Science; Vol.757*, pages 406–420. Springer-Verlag, 1992.

[17] Hewlett-Packard Company. *C and C++ SoftBench User's Guide*, June 2000. `http://docs.hp.com/en/B6454-97413/B6454-97413.pdf`.

[18] IBM Research. Jinsight. `http://www.research.ibm.com/jinsight`.

[19] Jan Kneschke. Lighttpd, 2009. `http://www.lighttpd.net/`.

[20] P. Kulkarni, W. Zhao, S. Hines, D. Whalley, X. Yuan, R. van Engelen, K. Gallivan, J. Hiser, J. Davidson, B. Cai, M. Bailey, H. Moon, K. Cho, and Y. Paek. VISTA:VPO Interactive System for Tuning Applications. In *ACM Transactions on Embedded Computing Systems (TECS)*, New York, New York, November 2006.

[21] N. Kumar, J. Misurda, B. R. Childers, and M. L. Soffa. Instrumentation in software dynamic translators for self managed systems. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, 2004.

[22] Y.-C. Lai, A. Motter, T. Nishilawa, K. Park, and L. Zhao. Cascade-based attacks on complex networks. *Pramana*, pages 483–502, 2007.

[23] D. B. Lange and Y. Nakamura. Program Explorer: a program visualizer for C++. In *COOTS'95: Proceedings of the USENIX Conference on Object-Oriented Technologies on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 4–4, Berkeley, CA, USA, 1995. USENIX Association.

[24] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. In *Software–Practice & Experience*, 1994.

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

[26] Microsoft Corporation. Visio 2007. http://office.microsoft.com/en-us/visio/default.aspx.

[27] S. Microsystems. The Awt in 1.0 and 1,1. Technical report, Sun Microsystems, April 1999. http://java.sun.com/products/jdk/awt.

[28] G. Necula. Cil - infrastructure for c program analysis and transformation, 2007. http://manju.cs.berkeley.edu/cil.

[29] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167, 2003. http://www.citebase.org/abstract?id=oai:arXiv.org:cond-mat/0303516.

[30] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 147–155, 1999.

[31] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[32] W. D. Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 116–134, London, UK, 1999. Springer-Verlag.

[33] PostgreSQL Global Development Team. PostgreSQL. http://www.postgresql.org, 2003.

[34] Rational Software. Rational Rose. http://www-01.ibm.com/software/rational.

[35] Red Hat. Red hat magazine, 2009. http://magazine.redhat.com.

[36] B. Starynkevitch. Compared gcc compilation time on two linux desktops. www.starynkevitch.net/Basile/compare_time_gcc.html.

[37] Sun Microsystems. Javadoc tool, 2004. http://java.sun.com/j2se/javadoc.

[38] Sun Microsystems, Inc. *dbx man page*. Sun Studio 11 Man Pages, Section 1.

[39] The CentOS Development team. Centos, 2009. http://www.centos.org/.

[40] The Eclipse Foundation. Eclipse. http://www.eclipse.org.

[41] S. R. Vegdahl. The Design of an Interactive Compiler for Optimizing Microprograms. In *Proceedings of the 18th annual workshop on Microprogramming*, December 1985.

[42] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 271–283, New York, NY, USA, 1998. ACM.

[43] Zephyr. Very portable optimizer, 1998. http://www.cs.virginia.edu/zephyr/vpo.