Proceedings of the
GCC Developers’ Summit

June 8th–10th, 2009
Polytechnique, Montréal,
Quebec, Canada
## Contents

The Visual Development of GCC Plug-ins with GDE  
*Dean, Callanan, Zadok*  
11

Collective Tuning Initiative  
*Grigori Fursin*  
31

Optimization opportunities based on the polyhedral model in GRAPHITE  
*T. Grosser*  
33

GIMPLE alias improvements for GCC 4.5  
*Richard Guenther*  
47

Interprocedural optimizations of function parameters  
*Martin Jambor*  
57

Using Eclipse for Reverse, Multi-Process and Non-Stop Debugging with GDB  
*Marc Khouzam*  
65

Adding named address space support to the GCC compiler  
*Michael Meissner*  
67

Hackers are from Mars and Corporations are from Venus  
*Michael Meissner and David Edelsohn*  
75

Automatic Streamization in GCC  
*Antoniu Pop, Sebastian Pop, Jan Sjödin*  
91

GDB Tracepoints, Redux  
*Stan Shebs*  
105

Design of Graphite and the Polyhedral Compilation Package  
*Jan Sjödin, Sebastian Pop, Harsha Jagasia, Tobias Grosser, Antoniu Pop*  
113
Porting GCC to Exposed Pipeline VLIW Processors
Alexandru Turjan, Dmitry Cheresiz, Roel Trienekens

Hybrid multi-architecture debugging with GDB
Ulrich Weigand
Conference Organizers

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

Review Committee

Ben Elliston, *IBM*
Andrew J. Hutton, *Steamballoon*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Google*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.
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

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 parallelism 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.

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 pre-processed 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_ADDRESS flag, which indicates an item can be passed to the run-time. This tree-like intermediate representation is called a parse tree and is what GCC uses, in various forms, throughout the compilation process.

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

![Diagram](image-url)
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.

![A subsection of a call graph rendered by GDE. Each node represents a particular function while edges represent function calls.](image)

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 non-trivial 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.

![Plug-ins to load](image)

Figure 4: A figure showing the plug-in loading process.
A user simply includes the flag `-fpre-process=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.

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](image)

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 allows 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 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
The Visual Development of GCC Plug-ins with GDE

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.

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 10: The GCC calling process. Actual file compilation and linking are done by files called by the gcc compiler driver.](image)

![Figure 11: The GDB Console of GDE.](image)
sults 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: Lighttpd [19] and the Linux kernel [39]. Lighttpd 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 1 and Table 2 show our dump statistics. 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).

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

![Size vs Statements](diagram.png)

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

![Diagram showing database size vs. number of statements](image)

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

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.

**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 dbdump 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 \texttt{COND_EXPR} node. Further, we were able to see that the first operand of a \texttt{COND_EXPR} (or conditional expression) node was the actual conditional test itself (in this case, an \texttt{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 \texttt{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.

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 \texttt{CALL_EXPR} node with several subtrees, the
Figure 14: Using GDE to get information about a COND_EXPR.

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.

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.

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 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
The Visual Development of GCC Plug-ins with GDE

of a mouse. For example, if a developer wanted to create and insert a new \texttt{COND_EXPR} node into a particular basic block, the developer would have to construct the \texttt{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 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 \textit{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.

**xpodb and VISTA:** *xpodb* 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 *xpodb* 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 of the *vpo* and designed to allow performance tuning of applications [20]. *VISTA* allows the user to step through transformation as *xpodb* 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 *xpodb* 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 kernel 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 straightforward 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.
Acknowledgments

Justin Seyster for his comments on an early draft of the thesis. Sean Callanan designed and implemented the GCC plug-in system.

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).

References


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


Abstract

Tuning default compiler optimization heuristic or optimizing programs, libraries and OS on a wide range of complex architectures is often a tedious, repetitive, isolated and time consuming task. We have developed a common web-based optimization repository (http://ctuning.org/cdatabase) to enable automatic and manual sharing of useful knowledge and experience about program optimizations from multiple users. We hope that it will automate and simplify compiler and architecture design as well as program and system optimization, and improve the quality of performance evaluation and benchmarking. It should help end-users and companies optimize their computing systems collaboratively using similar optimization cases from the community or using standard statistical and machine learning techniques to systematize a large amount of optimization data.
Abstract

The polytope model is used since many years to describe standard loop optimizations like blocking, interchange or fusion, but also advanced memory access optimizations and automatic parallelization. Its exact mathematical description of memory accesses and loop iterations allows to concentrate on the optimization problem and to take advantage of professional problem solving tools developed for operational research.

Up to today the polytope model was limited to research compilers or source to source transformations. Graphite generates a polytope description of all programs compiled by the gcc. Therefore polytope optimization techniques are not limited anymore to hand selected code pieces, but can actually be applied in large scale on real world programs. By showing the impact of GRAPHITE on important benchmarks - “How much runtime is actually spent in code, that can be optimized by polytope optimization techniques?” - we invite people to base their current polytope research on GRAPHITE to make these optimizations available to the large set of gcc compiled applications.

1 Motivation

The polytope model describes memory access optimizations based on an abstract mathematical representation. It can be used to describe traditional loop optimizations like blocking, tiling, or splitting in an exact way and to schedule them in arbitrary order. However there are also advanced auto parallelization\(^1\) passes and new optimizations based on powerful operational research tools, that cannot be expressed easily with traditional loop transformations, possible.

As the polytope model was always limited to source to source translation tools or hand selected code pieces, it was never used on regular base to optimize programs written in imperative languages. Therefore the big portion of C, C++ and Fortran programs was never accessible to optimizations based on the polytope model. GRAPHITE\(^2\) is the first open source implementation of the polytope model for a low level imperative compiler. GCC 4.4 already includes a first implementation of GRAPHITE, which is still limited in several aspects. It is tightly connected to gcc and the polytope description is not yet complete. During the last six months the current graphite branch was reworked and a complete polytope description was introduced. The last but very important thing missing is a set of advanced optimizations based on GRAPHITE and the polytope model. Here we can take advantage of the research taking part in this area since a long time.

Before starting to write/port new optimizations it is time to take a step back and see what can be optimized by GRAPHITE and if it is actually worth to write optimizations for.

How much code can be translated into the polytope model? How many conditions, statements, and loops of the original program are covered by the polytope description? How much runtime is actually spent in code that can be described and therefore optimized? And finally: If some code is not handled at the moment, is this just not yet implemented in GRAPHITE, is it a limitation because of missing optimizations in GCC or a limitation of the polytope model?

To find an answer, valid for a large set of programs, the SPEC 2006 benchmark suite is used to analyze the impact of GRAPHITE on “real world” programs.

\(^1\)http://www.fmi.uni-passau.de/cl/loopo/

\(^2\)http://gcc.gnu.org/wiki/Graphite
2 Status of GRAPHITE development

GRAPHITE is developed since several years. The first analysis, the scalar evolution pass, on which it is heavily based was committed five years ago. Last year the first public visible code was committed to gcc 4.4, released in spring 2009. However until today the impressive gains are still missing. So what is the development status? What is still coming and what does already exist?

To start there are two different versions of GRAPHITE. The version as in gcc 4.4 allows a simple transformation - loop blocking - to give a first impression of GRAPHITE and to test integration of CLooG and PPL. However GRAPHITE 4.4 is still very limited. Loop blocking does not have any heuristics at all and is based on a non polytope dependency analyzes from the lambda framework. This limits the effects and prevents big performance improvements.

The second version is in GRAPHITE branch and already took the next step. During the last 6 month the polytope description was completed, so that all future optimizations can take advantage of it and are not limited by any leftover code. Also all non polytope code was removed. This means in GRAPHITE branch there are currently no optimizations at all. However it is ready to base optimizations on it.

This paper will describe the situation in GRAPHITE branch, as this is the first full polytope model in gcc. In branch front end and back end, converting from GIMPLE to the GRAPHITE polytope description (GPOLY) and back from GPOLY to GIMPLE, work. This means we can extract the polytope description for interesting code regions and generate completely new loop nests from this description. This was tested by enabling the identity transformation \( \text{GIMPLE} \rightarrow \text{GPOLY} \rightarrow \text{GIMPLE} \) in gcc bootstrap and by testing large projects as the SPEC 2006 benchmark suite. Fortunately the missing part is the most interesting. Adding optimizations that work on the polytope representation.

There are two student projects about optimizations in GRAPHITE during Google Summer of Code \( ^{\text{TM}} \) 2009. The first one works on the already existing tree-autopar code for automatic loop parallelization with OpenMP. This code will be extended to handle all code graphite can optimize.

The other project aims to implement traditional loop transformations in GRAPHITE. Even if we already have loop-blocking in GCC 4.4, to actually take advantage of the polytope model and to get real performance improvements we need loop transformations that work completely on the polytope model and that are accompanied by well tuned heuristics.

The last missing part blocking both projects, the polyhedral data dependency analysis, was recently finished. Beside these two projects there are several other open projects, whereas most of them can take advantage of research already done. The LooPo [5] project has a large set of tools to discover more parallelism, there is research on iterative compilation or the internal vectorizer.
might want to take advantage of the exact dependency analysis. It is also possible to write new optimizations that use characteristics of the polytope model to optimize cost functions like the distance between memory access using the linear programming solver in PPL.

Another orthogonal project in GRAPHITE is PCP\(^3\). It was started half a year ago and will move GRAPHITE in a simple to handle package that includes a clean text and library interface for improved debugging and testing facilities. This should make development of GRAPHITE optimization passes even easier. For further information about the design of GRAPHITE and PCP is in the paper [7] available.

3 The polytope model

The polytope model is the model on which future GRAPHITE based optimizations will work. As there exists already a long research history there are several papers that talk about the polytope model [6], however most of them base the description on an source to source compiler.

In the context of modern compiler development this paper tries to offer a different introduction to the polytope model.

3.1 SSA on arrays - The polytope model?

Most modern optimizing compilers use an intermediate language based on static single assignment (SSA[4]) in their analysis and optimization passes. GIMPLE, the intermediate language used in GCC, is also based on SSA. The main reason for the use of SSA are the explicit use-def chains. Using SSA the lookup of the defining statement for a given use is in \(O(1)\). Furthermore for every use there is only one defining statement, therefore algorithms on SSA are often a lot simpler than on non SSA code.

Unfortunately SSA is only well defined on scalar variables (scalar SSA), whereas it is difficult to define a practical version of SSA on non scalar variables like arrays or memory references (array SSA). Even though GCC defines some kind of array SSA using “VUSE” statements, the implementation is limited compared to scalar SSA. In scalar SSA for every use of a scalar its current value depends on exactly one defining statement, as every write defines a scalar completely. In the array SSA implementation of gcc the content of an array can be defined by several statements, as a write does not define the complete array but can define single array cells. So the array may contain cells defined by different statements. Therefore there is a set of defining statements, not just one defining statement.

Making array SSA in the general case as useful as scalar SSA is a difficult problem, maybe impossible. However it is possible to represent a subset of programs using the polytope model, which can be seen as some kind of array SSA that extends the current “VUSE” statements.

3.1.1 Arrays as sets of scalar values

Even if it is difficult to define SSA on arrays, it is possible to extend scalar SSA. As arrays are defined as sets of scalar values, instead of tracking accesses to the complete array, accesses to every single array cell can be tracked. And for scalar values, SSA is already defined and useful. Unfortunately in the general case the array cells a statement accesses are not known at compile time, as the array subscript can be calculated by a function - in the worst case by a call to \(\text{rand}()\). So tracking them is not always possible.

Nevertheless there is a subset of array accesses that can be tracked exactly. All array accesses with compile time constant subscripts like \(A[5][8]\) or \(B[3][2]\). In this subset memory accesses to scalars and arrays can be defined in the same way. A memory access is the access to a scalar cell in the memory \(M\) of the program, a vector space in \(\mathbb{Z}^a\), whereas the first dimension defines the base element and further dimension define possible array subscripts. Scalar values like \(c\) are handled as arrays of dimension zero. To sort variables in this model we define a function \(f\) that assigns every scalar name and every array base an unique number \(i \in \mathbb{Z}\).

Now a memory access is not defined by its name, but as access to a cell \(c \in M\). With \(f\) defined as \({\{A \rightarrow 0, B \rightarrow 1, c \rightarrow 2\}}\) \(A[5][8]\) is the access to \((0,5,8)\), \(B[3][2]\) is the access to \((1,3,2)\), and \(c\) is \((2, 0, 0)\) as in 3.

By accessing array cells with this notation they are handled like normal scalars. So every use has exactly one defining statement, as every write defines a cell completely. However this is just the first part of SSA. Finding the defining statement is still a problem, as accesses to array cells are not yet in SSA. There may be several statements that write into \((0,5,8)\).

\(^3\)http://gcc.gnu.org/wiki/PCP
A[5][8] = 10;
A[5][8] = c;

for (i = 0; i < 100; i++) {
    for (j = 0; j < 50; j++)
}

for (i = 0; i < 100; i++) {
    (1,3,2) = (0,5,8); // S1
    (0,5,8) = (0,(2,0,0)); // S2
    (1,3,2) = (2,(0,5,8)); // S1
    (0,5,8) = (2,0,0); // S2

divide
}

for (j = 0; j < 50; j++)
    (0,3,4) = (3.1,(1,3,2)); // S3.1
    (1,5,8) = (3.2,1,(0,3,4)); // S3.2
}

3.2 Using a schedule to specify the defining statement

To identify a certain definition, the memory cell it accesses is not enough. Scalar SSA solves this by allowing only one definition of every scalar variable. Further writes into the same variable are forbidden. By adding this restriction the name of a scalar variable is enough to identify the defining statement. However for the scalar notation that was introduced on arrays a different approach is taken. Statements are referenced using a schedule. Every statement gets assigned an vector depending on its textual position in the program. The first statement is S:1, the second statement is S:2, ... . For every loop level an additional dimension is added that is incremented instead of the outer dimensions. If a new loop is for example started at position 3 the first statement in this loop has the schedule S:3.1. To specify the last definition of a use a tuple of schedule and memory cell is used. For example to reference the definition of memory cell (0,5,8) that took place in S:2, the tuple (2,(0,5,8)) is used as seen in 4.

Now every use can reference the single definition it is based on. This references can be used like normal SSA, but instead of a single assignment for every name there is a single assignment for every tuple (schedule, memory cell).

3.3 Extending array SSA for loops

The subset we used to define SSA on arrays is still very limited and is not able to represent a lot of real world code. Especially in loops it is uncommon to find constant array subscripts, as loops are often used to rework the content of a complete array. Fortunately it is possible to extend the defined array SSA.

To be able to represent arrays as set of scalars it was necessary to specify for every array access the accessed cell. This is easy for constant subscripts, but in general it is possible for all access functions that can be expressed and analyzed at compile time. Therefore array accesses like A[(i*M)^2] can be analyzed, as long as there is a way to store and analyze the function (i*M)^2 reasonably fast.

Unfortunately analyzing arbitrary functions is complex. However there is a subset of functions that can be analyzed with limited complexity. The set of functions, that are affine in virtual loop iterators (VLI) and parameters (P). Parameters are defined as scalar integer variables, that are constant during the execution of a code region expressed using polyhedral array SSA. Virtual loop iterators count the number of loop iterations. Therefore valid array accesses are all accesses, where the subscripts accessed are defined by an affine function like 5i + 8j + k + 12 as with i, j ∈ VLI and k ∈ P as shown in 5.

However by using this extension specifying the defining statement for an use is more complex. Scalar SSA and constant array SSA have the property, that in every loop the defining statement iteration for a memory
cell is always the previous iteration, as in every iteration the same memory cell is overwritten. If affine access functions are used, a statement can write in a different array cell on every loop iteration. Therefore a definition can only be referenced by the combination of the static schedule referencing a statement and a dynamic schedule specifying the loop iteration. In $S$ e.g. the last definition of memory cell $(1,25+k)$ was in statement $S3.1$ at iteration $(i=5)$. Therefore use-def chains are defined in between single statement iterations, instead of statements. As the number of statement iterations is prohibitive high, it is impossible to save all use-def relations for every single iteration. However it is possible to calculate these use-def chains as long as the expressions in loop boundaries and conditions are affine functions in $VLI$ and $P$. The usage of polytope libraries enables us to express the detailed use-def information in a highly compressed form. For $S4.1$ the affine function $j = 5i+k$ is used to identify the defs for every iteration of loop $j$. For iteration $j = 5+k$ the corresponding def is for example $t = 1$.

### 3.3.1 Why generate a new model and not extend GIMPLE?

It was shown that it is possible to extend SSA to arrays, so why is GRAPHITE not implemented as GIMPLE extension? One reason is that polytope array SSA duplicates a lot of information available in GIMPLE. Access functions, loop boundaries and conditional expressions are saved in the polytopes. Therefore all scalar code, except some reductions, is duplicated. Any transformation on the polytopes or the gimple code would require an expensive update of the other data structure. Also the polytope model is not expressive enough to be used for all GIMPLE memory accesses, so there would be two data structures to represent memory accesses. This complicates writing general algorithms. However the most important point is, that the polytope model is not imperative any more. Or at least it does not have to be imperative. A complete dependency description of the different statements and their iterations is enough to calculate the result of a SCoP. There is no need for control flow any more. So working on control flow based data structures like GIMPLE is too low level.

```c
void foo ( int k )
A [k] = c ;

for ( i = 0 ; i < 100 ; i ++ )

for ( j = 0 ; j < 100 ; j ++ )
... = B[ j ]
```

Figure 5: Affine array SSA

### 3.3.2 How to optimize polytope array SSA

There are two different ways to optimize in the polytope model. This first one is to change the order in which statements and loop iterations are executed. Most traditional loop optimizations try to optimize the use of the CPU cache, by either creating more cache locality using e.g. loop blocking or loop fusion or by reducing the memory footprint using loop splitting. All these transformations can be expressed by changing the execution order. However there are other use case to change the execution order. The two most interesting ones are transformations enabling parallel execution or vectorization.

Another orthogonal approach is optimizing the data layout. It is possible to reorder arrays to improve cache locality or to privatize data to allow further optimizations. It might even be possible to reduce the memory footprint by removing unused array regions or by removing temporary results.

### 3.4 Code GRAPHITE can represent

GRAPHITE calls a part of a program that can be represented in the polytope model a static control part
Optimization opportunities based on the polyhedral model in GRAPHITE

(SCoP). SCoPs are detected on the gcc intermediate representation GIMPLE, therefore all properties that qualify a code region as SCoP are independent of a specific language representation. Hence GRAPHITE is able to optimize programs written in any of the gcc front end languages (C, C++, Fortran, Ada, Java).

A SCoP is a single entry single exit region containing an arbitrary set of loop nests, straight line code and/or conditions, whereas the control flow has to be structured. However it is possible that the code in a SCoP is written using explicit for loops or implicit while, do..while or even goto loops. Loops that contain multiple exits are not supported.

Scalar variables that do not changed inside a SCoP are called parameters. Together with the loop induction variables the parameters span a vector space that is used to define affine expressions, whereas loops with multiple induction variables are supported.

In a SCoP only loops are allowed that have constant strides and which bounds can be expressed by affine expressions. Therefore unlimited infinite loops are not allowed. Conditions are limited to comparisons (<, <=, >, >=, ==, !=) between two affine expressions. However the same generality applies as for the control flow. Affine expressions do not have to be written explicitly in the source code, but it is sufficient that the expressions used can be analyzed to an affine expression.

Graphite handles all side effect free calculations in a SCoP. Thus even function calls are allowed if they are pure or const.

To be able to analyze dependencies efficiently data references to arrays are only allowed, if the subscripts are affine expressions, whereas graphite depends a lot on working alias analysis to be able to detect independent pointer sets. A SCoP can also contain scalar reductions as payload, they are handled like all data references. Structures can not be part of a SCoP.

SCoPs that meet these properties can be represented by the polytope model and can be optimized using generic polytope optimizations.

4 The polytope description in GRAPHITE

GRAPHITE implements a plain and simple interface\footnote{Source in “gcc/graphite-poly.h”} to optimize memory accesses based on the polytope model. As this model has been used for many years there exists already a well established description heavily influenced by ClooG\cite{2}, an open source polytope code generator.

The interface in GRAPHITE is close to the established descriptions used in polytope research, but contains some small differences and adjustments. Therefore it should be easy to port and try existing optimizations. As GRAPHITE’s polytope interface establishes a strict boundary to the GCC internal state. Analysis and optimization passes read, analyze and optimize an abstract mathematical problem description.

The small and independent interface allows people that are not yet part of the gcc community to try their polytope optimizations, without being forced to get used to gcc internals. A small header file with three data structures is enough to port a polytope optimization to the

```c
/* Start SCoP. */
start:
    A[i] = 1;
i++;
if (i <= 100)
goto start;
/* End SCoP. */

/* Start SCoP. */
void foo (int M, int N) {
    int i;
    for (i = 0; i <= M, i += 2)
    a = 10 * i + 12 + M;
    b = 5 * i + a;
    if (5 * i + 10 * M != a)
        A[b] = 20;
    else
        A[b] = b;
    /* End SCoP. */
}
```

Figure 6: Valid SCoP with goto-loop

Figure 7: Valid affine expressions in SCoP
void foo (int N, int M) {
    int i, j, r;
    int A[100];

    // Start SCoP
    A[N+1] = 0;   // bb0

    for (i = 0; i <= N+10; i++) {
        A[i+N] = A[i];   // bb1

        for (j = i; j <= i+M; j++)
            if (j > N)
                r = A[i-N] + r;   // bb2
    }
    // End SCoP
}

Figure 8: Example SCoP

GCC.
And if polytope optimizations once prove to be useful in compiler development, the simple interface is portable to a large set of open source compilers, so that a productive exchange and comparison of optimizations can take place.

4.1 The polytope library in GRAPHITE

To describe polytopes the Parma Polyhedra Library (PPL [11]) is used. It is the leading open source polytope library. For the import of GRAPHITE into gcc 4.4 PPL’s platform support was tested on and extended to all important gcc platforms.

All polytopes in GRAPHITE use the type

ppl_Pointset_Powerset_NNC_Polyhedron_t.

This type describes a union of non necessary closed convex polyhedra in the vector space $\mathbb{R}^n$. This is wrong as loop iterations and array accesses are elements of $\mathbb{Z}$, but works for the current uses. However to write more sophisticated optimizations, that require counting the points in the polyhedra, polyhedra in the vector space $\mathbb{Z}^n$ are required. So a switch to $\mathbb{Z}$-polyhedra will be necessary.

4.2 SCoP

A SCoP is a program region described in the polytope model. Therefore it only contains information concern-

ing the polytope model. Additional information describing the actual calculations is hidden. All optimizations and analysis take a SCoP, work on it, and return an optimized or analyzed version of it.

The SCoP $S = (p, bbs)$ is defined by the number of external parameters $p$ and a set of black boxes $bbs$.

A parameter is an integer variable that is unknown during compile time, but constant during SCoP execution. In the example $M$ and $N$ are parameters, as they are used in the SCoP, but not modified. $i$ and $r$ are no parameters as $i$ is an induction variable and $r$ is modified inside the SCoP. For this example $p = 2$.

4.3 Black Box

A black box describes a calculation that will be executed in the SCoP. As the name black box suggests the details of the calculation are hidden. In our example every statement is a black box on its own. Therefore a black box can contain a larger set of statements, some hidden control flow, or function calls. The only part exposed are the data references a black box contains.

The black box $B = (domain, drs, scattering)$ is defined by the iteration domain $domain$, a set of data references $drs$ and the scattering polytope $scattering$.

Every black box might be executed several times, whereas the loop induction variables are different for every iteration. A specific iteration of a black box is therefore referenced by the values of the loop induction variables for this iteration, whereas the possible values for the induction variables are the points in $domain$. $domain$ contains one dimension for every parameter of the SCoP and a special dimension representing constant offsets. In addition it contains one dimension for every loop iterator. E.g. $bb1$ has one loop dimension representing $i$, whereas $bb2$ has two dimensions for $i$ and $j$ and $bb0$ has no loop dimension at all. The domain for $bb$ looks like this 4.3

In contrast to other polytope optimizations packages the domain does not define the order in which different iterations are executed. Therefore it is not necessary for any optimization to change the domain.

The order in which the different iterations of a black box are executed is defined in the $scattering$ polyhedron. This polyhedron contains the same dimensions as $domain$, but is extended by several new dimensions $(t_1, t_2, ..., t_n)$ called scattering dimensions. The scattering polytope now maps every point of the domain to a point defined by the new scattering dimensions. For ex-
ample for \( bb1 \) the scattering function may define a mapping \( \{ t_1 = 0, t_2 = i, t_3 = 0 \} \) and for \( bb2 \) we may define a mapping \( \{ t_1 = 2, t_2 = i, t_3 = j \} \). To get the global execution order of the iterations of all black boxes the scattering dimensions are ordered lexicographically and the corresponding black box iterations are executed in this order. Therefore all iterations of \( bb1 \) will be executed before \( bb2 \) as \( t_1 < t_2 \) for all iterations. The individual loop iterations are executed in the original order, as for iteration \( i = 4 \) and \( i' = 5 \) there is \( t_2 < t_2' \).

Additional information about scattering functions can be found in the cloog documentation.\(^5\) However in contrast to cloog the scattering functions in G POLY are fully defined unions of non necessarily closed convex polyhedra. This allows to map subsets of the domain with different scattering functions.

\[ \text{Figure 9: Domain for black box } \text{bb2} \]

Data references in G POLY use an unified data model to describe the accessed memory. In classical GIMPLE there exist scalar values and arrays, whereas arrays are defined as a matrix of scalar cells. G POLY on the other hand talks about a multi dimensional space \( M \subseteq \mathbb{Z}^n \) of scalar values.

An access of an array cell \( A[s] \) is mapped to the point \( (d_1 = b, d_2 = s_1, d_3 = 0, \ldots) \), whereas the value \( b \) is defined by the alias set the array is pointing to. Every alias set is mapped to an unique value. If the array is part of more than one alias set every array cell is mapped to one point for every alias set the array is part of. The points only differ in the first dimension. Scalar values are handled like arrays of dimension 0.

In our example the read access of \( bb2 \) is defined as an access to \((2, 0)\) for the scalar access \( r \), and \((1, i - N)\) for the array access \( A[i - N] \). This can represented like in 4.4 as union of two polyhedra each containing one element.

At the moment data references are read only for optimizations, therefore the optimization of the data layout is not yet possible.

\[ \text{5http://www.bastoul.net/cloog/manual.php\#SEC8} \]

### 4.4 Data references

A data reference \( DR = (accesses, type) \), is defined by the accessed space \( accesses \) and the type of its access \( type \). The type of data reference can either be read, write or may-write. Read means a data reference reads or may read any of the values marked in accesses. Write means a data reference must overwrite all the values marked in accesses. May-write means that the values marked in accesses can be but do not need to be overwritten.

### 5 Coverage of GRAPHITE

After having seen which regions GRAPHITE can optimize it is time to ask how many SCopPs can be found in “real world” software. Is there a significant impact of GRAPHITE on code coverage or are there still restrictions left that limit coverage. In this analysis the SPEC
CPU 2006\(^6\) benchmark suite was used to provide an accepted set of test programs. Nevertheless the reader is asked to try GRAPHITE on its own programs.

5.1 How was the code coverage analyzed?

5.1.1 Compile time coverage

The coverage analysis in this paper is based on the idea of the “gcov” tool. The number of interesting items in the complete program is counted and compared to the number of touched parts of the program. However in contrast to “gcov” the analyze does not work on source code files, but on the gcc internal GIMPLE representation.

As the analyze is based on GIMPLE, not the lines of code are counted, but loops, conditions and statements in the GIMPLE control flow tree of the program. This simplifies comparing different front ends languages. Also the GIMPLE language is closer to the GRAPHITE optimizations, so a coverage analysis on it seems to be more related to the “real world” impact as an analysis based on source code.

Every item (loop, condition, statement) is interesting as soon as it can be detected as a part of a SCoP during GRAPHITEs SCoP detection, because future optimizations in GRAPHITE will be able to optimize it.

5.1.2 Hot spot coverage

As normally programs spend a lot of time in small parts of their code base, to foresee the performance impact of GRAPHITE a measurement that can model these hotspots is needed.

The model that is used is an extension to the GIMPLE based compile time coverage. It takes advantage of gccs feedback analysis “-fprofile-arcs -ftest-coverage -fprofile-use”. All benchmarks are run once with profiling instrumentation to get the execution count of every basic block. With the help of “-fprofile-use” this execution count is available for every basic block in gcc and was used to scale the loop, condition and statement counts depending on the number of times they where executed in the profiled test run.

These scaled counts will not give an approximation of the run time spent in a part of a program, as unoptimized hotspots can slow down program execution a lot. Nevertheless they give a better impression of the hot spot coverage and as they are independent of the processor architecture they can be used during GRAPHITE development to track progress in code coverage.

To get a accurate performance analysis for a specific test case on a specific hardware the usage of low impact performance measurement technics like PAPI might be usefull. However the SCoPs graphite detects are often very small so even little side effects might influence the measurement significantly.

5.2 Coverage of GRAPHITE branch - May 2009

At the moment (May 2009) GRAPHITE branch can handle most of the basic loop nests it was ment to handle, however it was never optimized for code coverage. Nevertheless it is important to get a start point to be able to see future improvements in code coverage.

\[eq\]
\[d_1/\text{first subscript}\]
\[e:
A \rightarrow 1
r \rightarrow 2
\]

Figure 10: Access space for read data reference of bb2
### Optimization opportunities based on the polyhedral model in GRAPHITE

#### Static Hotspot Benchmark

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>loops</th>
<th>conds</th>
<th>stmts</th>
<th>loops</th>
<th>conds</th>
<th>stmts</th>
</tr>
</thead>
<tbody>
<tr>
<td>459: GemsFDTD</td>
<td>2.24</td>
<td>0.71</td>
<td>0.57</td>
<td>0.28</td>
<td>0.27</td>
<td>0.07</td>
</tr>
<tr>
<td>473: astar</td>
<td>3.31</td>
<td>0.79</td>
<td>0.47</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>410: bwaves</td>
<td>3.45</td>
<td>1.35</td>
<td>1.46</td>
<td>0.19</td>
<td>0.16</td>
<td>0.09</td>
</tr>
<tr>
<td>401: bzip2</td>
<td>3.37</td>
<td>1.20</td>
<td>0.90</td>
<td>0.53</td>
<td>0.18</td>
<td>0.23</td>
</tr>
<tr>
<td>436: cactusADM</td>
<td><strong>2.50</strong></td>
<td><strong>0.96</strong></td>
<td><strong>3.54</strong></td>
<td><strong>81.83</strong></td>
<td><strong>68.83</strong></td>
<td><strong>99.41</strong></td>
</tr>
<tr>
<td>454: calculix</td>
<td>10.68</td>
<td>4.10</td>
<td>5.28</td>
<td>61.79</td>
<td>55.66</td>
<td>76.09</td>
</tr>
<tr>
<td>403: gcc</td>
<td>2.87</td>
<td>0.40</td>
<td>0.48</td>
<td>0.51</td>
<td>0.25</td>
<td>0.48</td>
</tr>
<tr>
<td>445: gobmk</td>
<td>2.54</td>
<td>0.41</td>
<td>0.51</td>
<td>0.30</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>435: gromacs</td>
<td>9.08</td>
<td>3.49</td>
<td>2.77</td>
<td>7.75</td>
<td>2.94</td>
<td>2.20</td>
</tr>
<tr>
<td>464: h264ref</td>
<td><strong>6.24</strong></td>
<td><strong>2.05</strong></td>
<td><strong>2.21</strong></td>
<td><strong>20.42</strong></td>
<td><strong>12.58</strong></td>
<td><strong>7.96</strong></td>
</tr>
<tr>
<td>456: hmmer</td>
<td>0.79</td>
<td>0.26</td>
<td>0.22</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>470: lbm</td>
<td><strong>19.23</strong></td>
<td><strong>7.35</strong></td>
<td><strong>35.79</strong></td>
<td><strong>0.07</strong></td>
<td><strong>0.02</strong></td>
<td><strong>0.02</strong></td>
</tr>
<tr>
<td>437: leslie3d</td>
<td>5.76</td>
<td>2.49</td>
<td>1.23</td>
<td>4.36</td>
<td>4.34</td>
<td>0.72</td>
</tr>
<tr>
<td>429: mcf</td>
<td>1.54</td>
<td>0.42</td>
<td>0.41</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>444: namd</td>
<td>0.16</td>
<td>0.03</td>
<td>0.05</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>471: omnetpp</td>
<td>0.96</td>
<td>0.06</td>
<td>0.08</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>400: perlbench</td>
<td>2.51</td>
<td>0.25</td>
<td>0.70</td>
<td>1.01</td>
<td>0.16</td>
<td>0.79</td>
</tr>
<tr>
<td>453: povray</td>
<td>3.57</td>
<td>0.75</td>
<td>0.69</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>458: sjeng</td>
<td>2.60</td>
<td>0.29</td>
<td>0.27</td>
<td>2.40</td>
<td>0.67</td>
<td>0.69</td>
</tr>
<tr>
<td>450: soplex</td>
<td>0.58</td>
<td>0.12</td>
<td>0.10</td>
<td>0.61</td>
<td>0.32</td>
<td>0.38</td>
</tr>
<tr>
<td>481: wrf</td>
<td><strong>18.57</strong></td>
<td><strong>6.36</strong></td>
<td><strong>7.70</strong></td>
<td><strong>34.45</strong></td>
<td><strong>29.72</strong></td>
<td><strong>54.86</strong></td>
</tr>
<tr>
<td>483: xalancbmk</td>
<td>1.18</td>
<td>0.19</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>434: zeusmp</td>
<td>3.41</td>
<td>1.03</td>
<td>1.11</td>
<td>0.84</td>
<td>0.43</td>
<td>0.11</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td><strong>4.66</strong></td>
<td><strong>1.52</strong></td>
<td><strong>2.90</strong></td>
<td><strong>9.45</strong></td>
<td><strong>7.68</strong></td>
<td><strong>10.62</strong></td>
</tr>
</tbody>
</table>

Figure 11: Coverage of GRAPHITE branch - May 2009 [in %]

Beside from this it is also good to see where are the low hanging fruits to improve the impact of GRAPHITE. As shown in 11 GRAPHITE achieves in average 4.66 % coverage of all loops and about 2.90 % of all SIMPLE statements. As expected the hot spot analysis shows that most of the loops are part of hotspots as 9.45 % of all loop iterations are covered by graphite and 10.62 % of all statement executions. Very interesting is the high divergency between the different results. There exist only four interesting benchmarks looking at the hot spot coverage. These are cactusADM with 99.41 %, calculix with 76.09 %, and wrt with 54.86 % statement coverage and h264ref with 20.47 % loop coverage. All other benchmarks are under 5 % most of them even less.

Looking at the compile time coverage the diversity is still big, but not that extreme. All benchmarks are in between 0.10 and 36.38 % statement coverage.

What is interesting is that a high value in compile time coverage does not imply high runtime coverage. Looking at cactusAMD it seems 3.64 % of all statements seem to be sufficient to get 99.41 % of the statement iterations. Whereas the 36.38 % statement coverage of lbm covers just 0.02 % of the executed statement iterations. Taking a look at the first coverage report for GRAPHITE it shows two things. On the one hand we already got some very interesting test cases for which it is worth to write optimizations for. On the other hand we learned that there are still a lot of benchmarks where GRAPHITE does not have much impact. However for a first shot the coverage seems promising enough to try to analyse the reasons for the small coverage in some of the benchmarks.

---

8loops, conds, stmts = simple coverage, whereas p-loops, p-conds, p-stmts is hot spot coverage
5.3 Ways to achieve better coverage

Until now focus in GRAPHITE development was to complete the polytope model by gathering necessary information from GIMPLE. As GPOLY is already complete enough to write optimizations on it, now it is time to focus on coverage. Good optimizations do not give any gains in performance, if they can not be applied.

5.3.1 What is possible in structured code?

There are different ways to improve coverage of GRAPHITE. To get an impression how much coverage can be achieved SCoP detection is run without any restrictions beside the structured control flow graph. Therefore it detects single entry single exit regions, without stopping on side effects, non affine loop bounds, structures, conditions that can not be handled or any other restrictions. They are only restricted to structured code containing only single exit loops and conditions with branches that can be joined easily. The analysis gives an idea of how much structured code can easily be accessed by GRAPHITE. As shown in 12 almost 50% of the statements and 64.31% of the statement iterations are in structured code.

However there is again a high diversity that shows e.g. statement coverage in between 11.25% and 90.50%. Certainly it is theoretically impossible to handle all of this, but GRAPHITE can move closer to these numbers.

5.3.2 Implementing missing features

The first step to extend coverage is to look inside GRAPHITE. There are several interesting features that are not yet implemented and can be added without extending GPOLY.

The first feature would be to handle arbitrary boolean expressions of affine functions in loop boundaries and conditions. At the moment we just support code like “SCoP 1” in 13, whereas it is possible to add all conditions shown in “SCoP 2” in the polytope model. Fortunately most of the work to simplify the conditions will be done by PPL. For normal conditions the only part left to be done is to the conditions into PPL and to see how fast PPL can simplify them. For loop bounderies it may be necessary to extend the analysis that detects the number of iterations for a loop.

```c
void foo (int N, int M) {
  int i , r ;
  int A [100] ;

  // Start SCoP 1
  for ( i = 0 ; i <= N + 10 ; i ++ ) {
    if ( 5 * i > i + N )
      r = A [ i - N ] + r ;
  }
  // End SCoP 1

  // Start SCoP 2
  for ( i = 0 ; i <= N + 10 || i <= M ; i ++ ) {
    if ( 5 * i > i + N 
        && ( i != 12 && N < M ) )
      r = A [ i - N ] + r ;
  }
  // End SCoP 2
}
```

Figure 13: Boolean expressions in bounds and conditions

It is also possible to support “may-write” for array accesses. Currently a memory access is rejected if the access function is not affine. However even it is not possible to represent the access function exactly, we can mark the complete region that they may be accessed as “may-write” or “may-read”.

Another way to extend coverage is to allow unstructured code. At the moment conditions are not allowed to be nested in complicated ways, loops with multiple exits can not be handled. A SCoP detection working on unstructured code should be able to handle this. Even if we can not represent unstructured parts of a SCoP in the polytope model, it might be possible to hide these parts inside a black box and treat them like an atomic operation. While working on more complex control flow graphs it would also possible to support `switch` statements.

5.3.3 Improve gcc analysis passes

GRAPHITE relies heavily on several gcc analysis passes. The most important ones are loop detection, scalar evolution and alias analysis.
As we have already seen in loop detection does a decent job.
However looking into the scalar evolution analysis there might still be room for improvement. Running SCoP detection without stopping for scalar evolutions that returned “scev-unknown”, shows that the executed statements increase from 10.62% to 22.63% as well as the other metrics. Therefore improving scalar evolution could have significant impact on GRAPHITEs code coverage. Another interesting pass is the alias and data reference analysis. This pass is not only interesting to achieve hight SCoP coverage, but also limits how much optimization is possible as less aliasing removes dependencies.

Beside these main passes all kind of interprocedural analysis are beneficial to GRAPHITE. Better constant propagation for example improves coverage as functions that contain a product of a parameter and another dimension become affine as soon as the parameter is known.

5.3.4 Extend the polyhedral model

It is also possible to extend the polyhedral model that is currently defined in GRAPHITE. One possibility is to allow conditions that can contain arbitrary expressions in their condition as long as the expressions do not touch any global state. Therefore the conditions are only allowed to read the values of loop induction variables or parameters. However allowing these conditions would require to keep track of them beside the polytope representation. Another possibility might be to extend the polytope model to handle parametric strides or array access functions containing non-linear parameters as described in [3]. However at the moment complexity seems to be prohibitive high. Therefore it seems to be necessary to wait for some optimizations that lower complexity at least for some special cases.

But even without extensions there is still enough space for improvement in GRAPHITE. So there is time for research and library development to work on tools and

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>loops</th>
<th>Static</th>
<th>Hotspot</th>
<th>loops</th>
<th>Static</th>
<th>Hotspot</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>conds</td>
<td>stmts</td>
<td></td>
<td>conds</td>
<td>stmts</td>
<td></td>
</tr>
<tr>
<td>459: GemsFD</td>
<td>96.55</td>
<td>60.80</td>
<td>68.24</td>
<td>100.00</td>
<td>99.98</td>
<td>100.00</td>
</tr>
<tr>
<td>473: astar</td>
<td>72.73</td>
<td>52.95</td>
<td>62.04</td>
<td>88.55</td>
<td>54.14</td>
<td>62.52</td>
</tr>
<tr>
<td>410: bwaves</td>
<td>97.70</td>
<td>82.88</td>
<td>81.07</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
</tr>
<tr>
<td>401: bzip2</td>
<td>53.37</td>
<td>37.75</td>
<td>43.26</td>
<td>54.82</td>
<td>27.05</td>
<td>30.74</td>
</tr>
<tr>
<td>436: cactus</td>
<td>72.81</td>
<td>48.08</td>
<td>62.92</td>
<td>99.33</td>
<td>98.48</td>
<td>99.96</td>
</tr>
<tr>
<td>454: calculix</td>
<td>77.31</td>
<td>52.97</td>
<td>59.11</td>
<td>97.34</td>
<td>92.89</td>
<td>97.67</td>
</tr>
<tr>
<td>403: gcc</td>
<td>43.29</td>
<td>15.63</td>
<td>19.66</td>
<td>63.17</td>
<td>39.57</td>
<td>46.28</td>
</tr>
<tr>
<td>445: gobmk</td>
<td>43.09</td>
<td>23.76</td>
<td>26.07</td>
<td>53.78</td>
<td>31.87</td>
<td>36.52</td>
</tr>
<tr>
<td>435: gromacs</td>
<td>83.91</td>
<td>61.00</td>
<td>73.57</td>
<td>57.96</td>
<td>36.38</td>
<td>83.44</td>
</tr>
<tr>
<td>464: h264ref</td>
<td>87.01</td>
<td>52.59</td>
<td>56.61</td>
<td>94.54</td>
<td>82.30</td>
<td>89.50</td>
</tr>
<tr>
<td>456: hmmmer</td>
<td>66.40</td>
<td>42.89</td>
<td>53.68</td>
<td>96.97</td>
<td>98.81</td>
<td>98.89</td>
</tr>
<tr>
<td>470: ibm</td>
<td>65.38</td>
<td>54.41</td>
<td>90.50</td>
<td>99.93</td>
<td>99.91</td>
<td>100.00</td>
</tr>
<tr>
<td>437: Leslie3d</td>
<td>96.64</td>
<td>56.09</td>
<td>74.40</td>
<td>99.93</td>
<td>99.79</td>
<td>99.98</td>
</tr>
<tr>
<td>429: mcf</td>
<td>27.69</td>
<td>18.99</td>
<td>19.44</td>
<td>14.76</td>
<td>12.53</td>
<td>10.69</td>
</tr>
<tr>
<td>444: namd</td>
<td>53.88</td>
<td>24.16</td>
<td>41.04</td>
<td>20.23</td>
<td>12.34</td>
<td>54.91</td>
</tr>
<tr>
<td>471: omnetpp</td>
<td>38.70</td>
<td>5.47</td>
<td>11.25</td>
<td>3.78</td>
<td>1.96</td>
<td>2.03</td>
</tr>
<tr>
<td>400: perlbench</td>
<td>40.26</td>
<td>13.17</td>
<td>16.49</td>
<td>4.57</td>
<td>11.81</td>
<td>18.33</td>
</tr>
<tr>
<td>453: povray</td>
<td>64.42</td>
<td>36.39</td>
<td>39.78</td>
<td>17.56</td>
<td>11.86</td>
<td>17.33</td>
</tr>
<tr>
<td>458: sjeng</td>
<td>61.69</td>
<td>21.91</td>
<td>23.60</td>
<td>58.75</td>
<td>36.39</td>
<td>35.44</td>
</tr>
<tr>
<td>450: soplex</td>
<td>61.57</td>
<td>37.53</td>
<td>44.37</td>
<td>86.91</td>
<td>78.86</td>
<td>78.96</td>
</tr>
<tr>
<td>481: wrf</td>
<td>92.05</td>
<td>67.45</td>
<td>63.42</td>
<td>99.00</td>
<td>97.53</td>
<td>98.43</td>
</tr>
<tr>
<td>483: xalanckmk</td>
<td>48.88</td>
<td>17.39</td>
<td>24.80</td>
<td>37.23</td>
<td>14.64</td>
<td>17.67</td>
</tr>
<tr>
<td>434: zeusmp</td>
<td>79.21</td>
<td>60.82</td>
<td>68.74</td>
<td>99.52</td>
<td>99.00</td>
<td>99.77</td>
</tr>
</tbody>
</table>

average | 66.28 | 41.09 | 48.87 | 67.33 | 58.18 | 64.31 |

Figure 12: Coverage on structured code without restrictions [in %]
algorithms for future GRAPHITE extensions.

6 Conclusion

After more than 30 years of research on polytope model based optimizations, with gcc 4.4 the first mainstream compiler was released that was able to apply polytope optimizations on low level imperative code. And this not only on small hand selected examples, but on all programs that can be parsed by the gcc. However the implementation is still very limited. During the last six months the remaining limitations were removed in GRAPHITE branch and by adding polytope data references the polytope model was completed. So the fundamentals are set to integrate polytope optimizations into gcc. Nevertheless there was a small part missing. It was not known how much impact GRAPHITE has. Does it actually make sense to write optimizations for GRAPHITE?

This paper shows that there are already 4 benchmarks in the set of 22 analysed SPEC benchmarks where GRAPHITE can extract enough SCoPs to have an noticeable impact without even optimizing it for high coverage. Therefore it seems possible to apply polytope optimizations in an automatic way on “real world” programs. The basis to start porting optimizations to GRAPHITE.

Furthermore several possibilities to extend code coverage in GRAPHITE were shown. On the one hand by extending GRAPHITE itself and on the other hand by improving the gcc analysis passes that are essential for GRAPHITE.

Taking this into account it seems realistic to expect future optimizations based on the polytope model in GRAPHITE to have notable impact. Therefore it is time to start working on both, improving GRAPHITEs coverage and writing optimizations that can take advantage of the model GRAPHITE exports.

References


Optimization opportunities based on the polyhedral model in GRAPHITE
GIMPLE alias improvements for GCC 4.5

Richard Guenther  
*Novell, SUSE Labs*  
gruenther@suse.de

**Abstract**

We present the evolution of alias analysis in GCC, introducing the new alias representation on GIMPLE and the GIMPLE alias-oracle. In particular we compare in detail the advantages and disadvantages of different levels of precision in the memory SSA form.

After introducing the GIMPLE alias-oracle future possible enhancements are outlined and the work is related to lowering the memory accesses in GIMPLE and unifying the RTL and GIMPLE alias-oracles.

**1 Introduction**

GCC, since the introduction of tree-SSA, uses a SSA representation for memory. With earlier versions this was the only available data-flow information for memory. The precision of that data-flow information was enhanced up to a point where its overhead is no longer manageable. With the most recent GCC release optimizers slowly migrate to using query-based disambiguators on top of the data-flow information present in the intermediate representation.

This paper presents the most recent developments in this effort targeting GCC 4.5. After giving an overview on SSA form for memory, the evolution of its implementation in GCC is outlined. Alongside several shortcomings are presented and the attempts to fix them are discussed. Finally the state of GCC 4.5 is presented and remaining challenges are outlined.

**2 Static Single Assignment Form**

Static Single Assignment (SSA) form provides a sparse data-flow representation for scalars on top of an infinitely large register file. A register in SSA form has exactly one definition. Multiple assignments to a scalar variable are translated to multiple definitions of separate registers (SSA names). An SSA name is identified by its base variable \( B \) and its version \( n, B_n \).

\[
\text{int } i = 1; \\
i = 2;
\]

gets translated to

\[
\text{int } i_1 = 1; \\
i_2 = 2;
\]

A SSA name that has no definition is called a default definition and is denoted as \( i_3(D) \). These get used for scalar function parameters where their definition is not available and in case of uses of uninitialized scalars in which case the definition is missing.

As there is only one definition for a register, SSA form needs a merge operator that factors the use-def chains at confluence nodes in the control-flow graph. This factoring also makes sure that the single definition of an SSA name dominates all its uses. The so called PHI nodes merge SSA names coming in from different edges into a basic-block.

\[
\text{if (b)} \\
\begin{align*}
  i &= 1; \\
  \text{else} \\
  i &= 2; \\
  j &= i;
\end{align*}
\]

gets translated to (omitting the condition and branch)

\[
<\text{bb 3}>: \\
i_1 = 1; \\
goto <\text{bb 5}>;
\]

\[
<\text{bb 4}>: \\
i_2 = 2;
\]

\[
<\text{bb 5}>: \\
\begin{align*}
  \# i_3 &= \text{PHI } i_1(3), i_2(4) \\
  j_4 &= i_3;
\end{align*}
\]

merging \( i_1 \) and \( i_2 \) and defining the merge result \( i_3 \). When rewriting the program out of SSA form the PHI nodes get replaced by copies on the edges.
With a program in SSA form the data-flow problem of reaching definitions is constant, the data-flow problem of reaching uses is linear. GCC implements these by keeping a link from the SSA name to its single definition statement and by maintaining a list of uses for each SSA name[2].

With memory there are partial definitions and uses and as memory can have its address taken accesses cannot be generally decomposed into scalar portions. This means that objects of aggregate types or memory operations involving pointers are not well suited to be represented by SSA form due to the single definition restriction. Consider rewriting $s$ in the following example into SSA form

```c
struct { int i; int j; } s;
s.i = 1;
s.j = 2;
foo (s);
```

where you need to be able to preserve part $s.i$ at the assignment to $s.j$ and have both parts available at the use of $s$. In simple cases like this you can work around the issue by explicitly preserving untouched parts using partial uses like

```c
struct { int i; int j; } s;
s_2 = { 1, s_1(D).j };  
s_3 = { s_2.i, 2 };  
foo (s_3);
```

but this obviously is not a cheap representation nor does it work if $s$ is stored to through a pointer. Though the above trick is commonly used to rewrite scalars of complex or vector type into SSA form.

### 3 Aliasing

Aliasing refers to the fact that a single memory location can be referred to by multiple lexically different designators. In C the simplest case is

```c
union { int i; float f; } u;
```

where $u.i$ and $u.f$ refer to the same memory location but are distinct designators.

Most aliasing occurs due to the use of pointers. Accesses through two pointers $p_1$ and $p_2$ are said to alias if they refer to the same memory or if one refers to a subset of the other referred memory.

For alias relationships between two memory designators $M_1$ and $M_2$ we distinguish

- $M_1$ may-alias $M_2$ if either $M_1$ or $M_2$ may refer to all or a part of the other
- $M_1$ must-alias $M_2$ if $M_1$ and $M_2$ refer to the same memory location

The alias relation $M_1$ may-alias $M_2$ is always conservatively correct. This conservative assertion limits most optimizations that affect memory loads or stores severely.

#### 3.1 Disambiguation

In the face of aliasing optimization passes often need to prove that $M_1$ and $M_2$ may not alias to assert that a transformation on the program is valid. For example dead store elimination needs to know which of the reaching may-definitions are killed by a store.

A memory designator can be decomposed into a base which is either an identifier or a pointer dereference, an access size and an access range consisting of an offset and an extent. Pointer analysis (points-to analysis, PTA) computes a set of identifiers a pointer may point to, its points-to set.

For disproving aliasing of $M_1$ and $M_2$ we distinguish memory designators based on pointer dereferences and memory designators based on identifiers.

- Two memory designators based on identifiers do not alias if the identifiers are not equal or if the accesses on equal identifiers do not overlap.
- Two memory designators based on the same pointer do not alias if the accesses do not overlap.
- A memory designator based on a pointer and a memory designator based on an identifier do not alias if the identifier is not included in the pointers points-to set.
- Two memory designators based on two different pointers do not alias if the intersection of their points-to sets is empty.

The first two cases are referred to as offset-based disambiguation, the last two cases are called pointer-based disambiguation.
In addition to offset-based and pointer-based disambiguation, some programming languages allow disambiguation with type-based alias analysis (TBAA). In 6.5/6 and /7 ISO C99 specifies which effective types of lvalues may possibly access the stored values of an object with an effective type.

Type-based disambiguation is useful whenever at least one pointer-based memory designator is present. There are two useful types in an access for TBAA.

- The type of the base of the memory designator constrains the access path origin. Accesses in object hierarchies can be disambiguated with it.
- In addition, the type of the access result allows disambiguation on simple accesses.

For the second case an example is that in C it is guaranteed that an access through a pointer of type int * does not alias an access through a pointer of type float *.

For the first case consider

```
struct Base { int i; int j; } base;
struct Deriv { struct Base b; int k; } *q;
```

where for base.i and q->b.i the types of the result are the same but due to the dereference *q that is of type Deriv we can disambiguate both accesses because the object base may not be accessed through a pointer to Deriv.

### 3.2 Function Calls

Function calls in general are points of may-definition and may-use for all escaped memory. Escaped memory includes all memory that a called function possibly can refer to, either directly or indirectly. A may-definition by a call is also referred to as call-clobber, a may-use by a call is also referred to as call-use.

Escape analysis, usually done as part of PTA, can be used to disambiguate memory designators against calls.

### 4 SSA Form for Memory

SSA form for memory tries to build a sparse data-flow representation for memory. This is data-flow information readily available to optimization passes so they do not have to concern themselves with the validity of transformations involving aliased references. SSA form for memory was first proposed by Chow[1].

SSA form for aliases has to deal with may-definitions and may-uses. To accommodate this we factor the use-def chains at the points of may-definitions in addition to confluence nodes in the CFG. This factoring is done by making the definition of an SSA name use the SSA name from the immediately dominating may-definition. This ensures there are no overlapping lifetimes of SSA names representing possibly the same memory content.

The constraints on SSA form for memory are so that from a memory store all may-defs are reachable when following the use-def chain of the SSA names representing memory state defined by the store and that all may-uses are reachable when following their def-use chains.

We introduce so called virtual definitions, VDEFS, and virtual uses, VUSES, forming virtual SSA form. The virtual SSA names represent the state of memory at their point of definition. Each virtual definition has an accompanied virtual use. The virtual definition

```
# a_3 = VDEF <a_2>
```

uses a_2 and defines a_3 invalidating the memory state associated with a_2. Virtual uses by means of the virtual SSA name use-def link link to the statement defining the memory state still valid before the use statement.

We assign a set of aliased memory to the virtual SSA name bases. This allows local re-building of the virtual definitions and uses. A consequence is that statements need to have virtual definitions and uses for all aliased memory to represent all possible conflicts correctly. In turn the virtual SSA representation is less sparse than one recording only locally necessary conflicts. The advantage is that it allows incremental updates that are necessary when moving or inserting statements with may-definitions. In Fig.1 you can see two SSA edges between the store to *p and *q while only one would be strictly necessary to represent the conflict between them.

The following illustrates SSA form on memory.
int i;
void foo(int *p, int *q)
{
    # i_2 = VDEF <i_1(D)>
    i = 0;
    # i_3 = VDEF <i_2>
    # SMT.1_5 = VDEF <SMT.1_4(D)>
    *p = 1;
    # i_6 = VDEF <i_3>
    # SMT.1_7 = VDEF <SMT.1_5>
    *q = 1;
}

Figure 1: Redundant conflict between *p and *q.

struct X { int i; int j; };
X a, b;
# a_2 = VDEF <a_1(D)>
    a.i = 1;
# a_3 = VDEF <a_2>
    a.j = 2;
# VUSE <a_3>
# b_5 = VDEF <b_4(D)>
    b.i = a.i;

Unlike for scalar SSA form the virtual definitions are may-definitions, the virtual uses are may-uses. For example a_3 refers to the store to a.j but is used by the read from a.i. Likewise the store to a.j invalidates the memory state represented by a_2, but that refers to the store to a.i.

Both reaching may-definitions and reaching may-uses are linear problems in virtual SSA form. Whether a may-def is a must-def, or a may-use is a must-use remains as a separate problem to solve.

GCC uses virtual SSA form as outlined before to represent data-flow of memory in the GIMPLE intermediate representation. Virtual SSA form was introduced at the same time that SSA form on trees was introduced, GCC 4.0. Implementation details, representation and constraints changed in every minor release throughout the present GCC 4.5 pre-release.

The following presents a brief history of that evolution.

GCC 4.0 and later versions use different kinds of so called alias-tags that are used as base for virtual SSA names.

- For declared objects the alias-tag is the identifier of the object.
- PTA introduces so called name-memory-tags, NMTs, that are used to represent the collection of pointed-to memory for a specific SSA name pointer. Each name-memory-tag records all possibly pointed-to identifiers as aliases. When anonymous memory is amongst the set of pointed-to memory NMTs are not used but instead SMTs are used for those pointers.
- Anonymous memory accessible by a pointer to a specific type is identified by so called symbol-memory-tags, SMTs (also known as type-memory-tags, TMTs). Each symbol-memory-tag records all aliased symbol-memory-tags and identifiers that are aliased according to TBA when dereferencing a pointer of the specific type.

4.1 Operand Scanner

In GCC the operand scanner is responsible for creating and removing virtual definitions and uses on statements. The operand scanner operates locally, thus the complete set of alias-tags representing may-defined and may-used memory need to be computable from looking at a single statement.

The set of alias-tags used is computed as follows. If the memory designator is based on an identifier the identifier is used as its single alias-tag. If the memory designator is based on a pointer and it has points-to information available and does not refer to anonymous memory, the set of pointed-to alias-tags recorded in its name-memory-tag is used. If the memory designator is based on a pointer and it has no points-to information available, its symbol-memory-tag and the set of aliased alias-tags recorded in it are used.

In Fig. 2 you can see all three cases, a plain store to i, a store to possibly anonymous memory using the symbol-memory-tag SMT.1 and its aliased identifiers i and j, and a store via a pointer with points-to information available, q, that aliases only i and j.

The set of aliased alias-tags for symbol-memory-tags is computed during TBA, that of name-memory-tags is computed during PTA. At the same time alias-tag sets for call-clobbered and call-used memory are put together. The smaller these sets the more precise and sparse is the data-flow information in the memory SSA IL and the less overhead is caused by it.
int i, j;
void foo(int b, int *p)
{
    int x, *q;
    if (b)
        q = &i;
    else
        q = &j;
    # i_5 = VDEF <i_1(D)>
    # j_6 = VDEF <j_2(D)>
    # SMT.1_4 = VDEF <SMT.1_3(D)>
    *p = 0;
    # VUSE <i_5>
    # VUSE <j_6>
    x = *q;
    # i_7 = VDEF <i_5>
    i = 1;
}

Figure 2: GCC virtual SSA form.

4.2 GCC 4.1

GCC 4.1 introduced a new kind of alias-tag, the structure-field-tag, SFT, that represents a part of a structure. Thus virtual SSA form became field-sensitive, increasing the precision of data-flow information and at the same time the sets of aliased symbols, memory-usage and compile-time.

struct X { int i; int j; };
X a, b;
# SFT.1_2 = VDEF <SFT.1_1(D)>
a.i = 1;
# SFT.2_4 = VDEF <SFT.2_3(D)>
a.j = 2;
# VUSE <SFT.1_2>
# SFT.3_6 = VDEF <SFT.3_5(D)>
b.i = a.i;

Compared to the same example before the use of SFT.1_2 now directly refers to the must-aliasing store to a.i as the unrelated store to a.j is now covered by the alias-tag SFT.2. This enables elimination of the redundant load from a.i propagating the constant 1 to the store of b.i.

The increase in precision made a big difference as on the tree level at that time no other memory disambiguations were available other than skipping unrelated statements via following the virtual use-def chains.

int i, j;
void foo(int b, int *p)
{
    int x, *q;
    if (b)
        q = &i;
    else
        q = &j;
    # MPT.1_2 = VDEF <MPT.1_1(D)>
    # SMT.1_4 = VDEF <SMT.1_3(D)>
    *p = 0;
    # VUSE <MPT.1_2>
    x = *q;
    # MPT.1_5 = VDEF <MPT.1_2>
    i = 1;
}

Figure 3: GCC virtual SSA form.

4.3 GCC 4.2

Type-memory-tags were renamed to symbol-memory-tags, SMTs. The concept of SFTs was extended to also cover array elements.

Again with the lack of memory disambiguations this made a noticeable difference for optimizing computations on small arrays as it often happens with C++ code for vector and matrix arithmetic.

4.4 GCC 4.3

It is important to keep the set of aliased tags small for both symbol-memory-tags and name-memory-tags as otherwise the size of the intermediate representation grows very large and optimization passes traversing the virtual use-def and def-use links consume a lot of time. Especially with the increased use of SFTs this became a big problem.

To help this situation the mem-SSA project[3] introduced the concept of alias-tag partitioning. With that multiple alias-tags are merged into a new representative, a memory-partition-tag, MPT, that is then used in place of multiple alias-tags reducing memory-usage, compile-time but also precision of the data-flow information.

In Fig.3 you can see the result of partitioning the alias-tags i and j to the new alias-tag MPT.1. Thus, partitioning decreases the data-flow precision while at the
same time reducing compile-time and memory-usage overhead of the virtual SSA form.

Partitioning is done by sorting alias-tags by the predicted frequency of their use and then partitioning alias-tags which alias (and thus are related) together starting with the least frequently used tags and stopping when a given threshold of remaining alias-tags is reached.

The goal is to keep the interesting bits, thus the precision, and reduce the noise, thus the overhead. Both partitioning related alias-tags in one partition and not partitioning a certain amount of most often used alias-tags tries to achieve this. The operand scanner constrains the partitioning to be the same for the whole function. Refining the partition in regions that are more interesting for optimizations is not possible.

One effect of partitioning is that once functions grow, for example by inlining, the precision of data-flow decreases and less optimizations are performed. This results in somewhat erratic performance results that are obtained when compiling with GCC 4.3. Removing or adding seemingly unrelated code to a function can change the optimization results on a different part of the function.

While a lot of compile-time and memory-usage regressions were fixed by doing the alias-tag partitioning new compile-time regressions came up that showed that the computing the partitioning itself can take up to 70% of the compile-time of a program.

4.5 GCC 4.4

To compensate for the loss of data-flow precision from the introduction of MPTs a general alias-oracle was introduced for use by tree optimizers. At the cost of compile-time the oracle walks the virtual use-def and def-use chains applying alias analysis to skip non-interesting statements.

This recovered most of the run-time regressions caused by introducing alias-tag partitioning while not regressing in compile-time or memory-usage again. The alias-oracle is only in effect for tree level full and partial redundancy elimination, and for loop invariant and store motion.

5 GCC 4.5 Virtual SSA Form

With GCC 4.5 the virtual SSA form will be as sparse as possible, reducing the overhead on top of the intermediate representation to a single alias-tag, .MEM, and at most one virtual use or one virtual definition per statement. That ties the data-flow information in virtual SSA form down to conservatively answering may-alias all the time so that all disambiguation has to be done using the tree alias oracle. The alias oracle was sufficiently enhanced to compensate the loss in precision.

The reasoning for this change is that keeping the overhead low of representing and maintaining the virtual SSA form at the same time degrades it to nearly the same level as the new form. Further even keeping the representation correct proved to be difficult and for this reason virtual SSA form was re-created from scratch several times during optimization. At the same time the original promise of providing a conservative default data-flow for memory to optimization passes is kept, simplifying the implementation of passes and more importantly ensuring a smooth transition to the new scheme.

One major obstacle for balancing overhead and precision of virtual SSA form is maintaining correct call-clobber and call-use information on call statements. The set of alias-tags necessary to represent call-clobbered variables is determined by transitively closed points-to solutions for escaped memory. Once that solution gets big or not constrainable (thus, the solution contains all address-taken variables) the set of alias-tags used or defined on call statements grows linear in the number of variables. Simply partitioning them into a single partition would reduce precision to zero for this unfortunately common situation.

Maintainability of the code that is necessary to build and relate all the alias-tags and to partition them as well as the difficulty to debug problems with that code is also a major reason to get rid of it. The number of bugs regarding to correctness of call-clobber handling and partitioning SFTs support this as well.

The simplified virtual SSA form for GCC 4.5 for the example in Fig. 2 is shown in Fig. 4. As you can see the overhead of the virtual SSA form shranked by 50%. The actual savings are bigger due to simplified data-structures used for maintaining virtual SSA form which also improves the compile-time for maintaining and using it further.
```c
int i, j;
void foo(int b, int *p)
{
    int x, *q;
    if (b)
        q = &i;
    else
        q = &j;
    # .MEM_2 = VDEF <.MEM_1(D)>
    *p = 0;
    # VUSE <.MEM_2>
    x = *q;
    # .MEM_3 = VDEF <.MEM_2>
    i = 1;
}
```

Figure 4: GCC 4.5 virtual SSA form.

6 GCC 4.5 Alias Oracles

For disambiguating memory references there are several query-based analyses, also called alias oracles, available. A building block for the more general oracles is the TBAA oracle. On top of that there are oracles for memory references in RTL form and for memory references in tree form.

In general optimization passes need to ask the correct questions to conform with the memory-model imposed by the active language. Standard dependence types that need to be distinguished are

- **read-dependence**, whether a read depends on a preceding read
- **true-dependence**, whether a read depends on a preceding write
- **anti-dependence**, whether a read conflicts with a following write
- **output-dependence**, whether a write conflicts with a following write

For answering anti-dependence and output-dependence tests GCC will not use TBAA to accommodate C and C++ type-based aliasing rules.

6.1 TBAA Oracle

Type-based alias analysis in GCC works by the front-ends and the middle-end jointly assigning alias-sets to types. An alias-set is identified by a unique identifier, the alias-set number, with which several types may be annotated. An alias set $A_2$ is a subset of the alias-set $A_1$ if all elements of $A_2$ are included in $A_1$ but not the other way around. The special alias-set number zero identifies the alias-set that is a superset of all alias-sets.

If an alias-set $A_1$ for type $T_1$ is a subset of the alias-set $A_2$ for type $T_2$ then $T_1$ may be used to access (part of) an object of type $T_2$.

Two alias-sets are said to conflict if their alias-set numbers are equal or if at least one alias-set is a subset of the other. It follows that all alias-sets conflict with alias set zero.

A memory designator based on a pointer dereference using type $T_1$ with alias-set $A_1$ conflicts with a memory designator based on a declaration of type $T_2$ with alias-set $A_2$ if $A_1$ is equal to or a subset of $A_2$.

Two memory designators based on pointer dereferences using types $T_1$ and $T_2$ with alias-sets $A_1$ and $A_2$ conflict if their alias sets conflict.

The TBAA oracle is queried by the two main entries alias_set_subset_of and alias_sets_conflict_p. The alias-set number for a type can be queried using the get_alias_set function.

6.2 RTL MEM-EXPR Oracle

The MEM-EXPR oracle does offset-based disambiguation on memory RTXen and provides classic dependence testing routines that combine the TBAA oracle with the offset-based disambiguations.

Offset-based disambiguation is done by nonoverlapping_memrefs_p, the dependence testing routines are read_dependence, true_dependence, anti_dependence and output_dependence.

6.3 Tree Oracle

A tree-level alias-oracle was introduced in GCC 4.4 consisting of the single entry-point refs_may_alias_p trying to disambiguate two memory reference trees based on TBAA and offset-based disambiguation.
For GCC 4.5 this main disambiguator was enhanced to also use the results of points-to analysis. Dependence testing routines are available as \texttt{refs\_anti\_dependent\_p} and \texttt{refs\_output\_dependent\_p}.

The tree-level alias-oracle relies on \texttt{get\_refs\_base\_and\_extent} to gather information on the size of the access as well as its offset and its maximum extent. For example the memory reference \texttt{x.a[i]} on

\begin{verbatim}
struct { int i; int a[8]; int j; } x;
\end{verbatim}

has a size of 32 bits (one int), an offset of 32 bits (the offset of a) and a maximum extent of 256 bits (the size of a). With this information we can disambiguate against \texttt{x.j}.

For the use of points-to analysis results these have been made easier to access by introducing an abstraction for a points-to solution, \texttt{struct pt\_solution} which is used for points-to sets for SSA name pointers and for the function-wide solutions for the set of call-used and escaped variables. The following query functions are available:

\begin{itemize}
  \item \texttt{pt\_solution\_includes\_global}, whether a points-to solution includes global variables
  \item \texttt{pt\_solution\_includes}, whether a points-to solution includes a specified variable
  \item \texttt{pt\_solutions\_intersect}, whether two points-to solutions intersect
\end{itemize}

The tree-level oracle adds access-path based disambiguation that relies on the availability of TBAA and uses offset-based disambiguation to query if of two access paths one is based on the other and thus two accesses may alias. Consider a situation that may happen quite often within C++ inheritance hierarchies:

\begin{verbatim}
struct Base { int i; int j; } *p;
struct Deriv { struct Base b; int k; } *q;
\end{verbatim}

To disambiguate \texttt{p->i} against \texttt{q->b.j} we need to realize that \texttt{p} must point to \texttt{q->b} to alias \texttt{*q}. Thus we search for a prefix in either reference that, together with the base of the other reference, can serve as a base for offset-based disambiguation, in this case \texttt{*p} and \texttt{q->b}. The comparison is based on type equality and is only valid if the program follows strict type-based aliasing rules of the language.

In the case that there is no common base the references can be trivially disambiguated as you cannot reach the same memory through access paths starting from different roots. For example \texttt{p->i} and \texttt{q->k} can be disambiguated that way.

### 6.4 Tree Statement Walkers

There are two general statement walkers available to walk may-aliased reaching definitions (\texttt{walk\_aliased\_vdefs}) and to walk not may-aliased reaching definitions (\texttt{walk\_nonaliased\_vuses}).

The first is used by dead code elimination to mark all reaching definitions of a load as necessary, the second is used by value-numbering to look up memory expressions in the hash tables for all equivalent memory states.

### 7 Present Challenges

The following presents the challenges GCC 4.5 faces with respect to alias analysis. Most of them should be addressed within the GCC 4.5 time-frame.

#### 7.1 IPA PTA

With LTO on its way inter-procedural points-to analysis will become reality. This puts a new constraint on local PTA such that it cannot be re-done after the IPA propagation phase.

This can be addressed by properly merging points-to solutions of different functions during inlining and by strengthening the propagation and merging of points-to solutions during the local optimization phases.

#### 7.2 Memory Model

Adjustments to the memory model used by the GCC middle-end are necessary to accommodate for C and C++ that allow the effective type of anonymous memory to change. This effectively will disallow TBAA based disambiguations for anti- and output-dependencies.
7.3 C restrict

The C language has the concept of restricted pointers that allows the programmer to specify that two memory designators based on two restrict qualified pointers that are not based on each other do not alias.

Presently restrict is implemented as part of the TBAA alias-set tree. This does not match its desired semantic constraints nor would a points-to based solution. The based-on concept is what makes a new points-to-like concept necessary.

7.4 Alias Export

Points-to information is currently not available to the RTL memory disambiguators. Exporting this information from the tree level to RTL level at expansion time is necessary to overcome the pessimization of the new memory model especially for scheduling.

7.5 LTO and TBAA

With LTO and especially with multi-language LTO applying type-based alias analysis has semantic issues. Different languages place different constraints on type-based analysis, and even within the same language but different translation units semantics are non-obvious.

The GIMPLE middle-end needs to provide type-based alias analysis based on a type system that conservatively accommodates all languages at the same time. Pure structural equivalence sounds like what it is going to be.

7.6 Merging the Oracles

The alias-export branch currently maintains the original tree memory references in the MEM attributes to be able to use the tree alias oracle. With instead storing the memory reference base, its offset and maximum extent the tree and RTL offset-based disambiguators could be unified possibly only losing the access path based disambiguations. This directly relates to ongoing efforts to lower the representation of memory accesses to a flat representation no longer recording access paths but plain base and offset information.

References


Abstract

The generic nature of large applications introduces many layers of abstraction made of functions which are essentially passed information that they do not need or in a way that is more complex than necessary. The paper describes a new early interprocedural pass that deals with these redundancies and genericalness by avoiding unused parameters, transforming parameters that are passed by reference to ones passed by value when legal and passing only a portion of an object when beneficial and thus trying to avoid the penalty and regularize calls. It discusses its objectives, interesting implementation details and issues, how the pass fits into the rest of the framework of interprocedural optimizations and a number of benchmark results on C++ applications. A substantial part of the paper describes the new implementation of intraprocedural SRA with which the interprocedural one shares a lot of code.

1 Introduction

All bigger programs essentially use a number of layers of abstraction. At the very least, related data are stored together in chunks, usually called objects, and associated functions operate on those in a black-box manner to facilitate isolation of interface and implementation. All functions then usually take pointers to such objects as their parameter, even if they operate only on a part of the object, sometimes not even modifying it. Thus they become a barrier to optimizations and the address of the object is considered escaped, preventing further optimizations of accesses to the object (most notably intraprocedural SRA [9]). Moreover, there are other cases when the original function parameters in a program are inefficient. Generic techniques of programming can also lead to parameters which are not used at all.

GCC deals with these obstacles in a number of ways. It classifies functions as pure or const based on whether

---

Figure 1: Effect of relaxing inlining limits on compile time of Tramp3D [2] at -O2 with early inlining turned off.

and how they access memory [5]. This analysis is useful in many ways but the granularity of this approach does not differentiate in between dereferencing a parameter and other memory accesses and characterizes the whole function, not individual parameters. Alternatively, compilers can employ some sort of address escape analysis [6] to assert that an address passed to a function is not stored to memory where it cannot be tracked and to relate such addresses together. Escape analysis does not help alone, though, and other parts of the compiler have to learn to use it.

Finally, by far the most powerful means of overcoming these barriers is inlining [9, 3, 4, 5], especially when all functions that operate on an object are inlined. GCC inlines heavily, so that even programs such as Tramp3D [2] with multitudes of small functions are optimized thoroughly. Yet excessive inlining does not scale indefinitely and causes huge increases in binary and compile time (see figure 1). Severity of these problems increases with the compilation unit size and will certainly be an issue with LTO.
2 Regularization of calls

Given that the parameters which result from high-level and generic programming techniques have such an impact even when not strictly necessary, our conclusion was to try to convert function parameters to the least harmful form possible. In particular to:

1. Remove parameters that are not used by the function.
2. Convert parameters that are passed by reference to ones passed by value if possible.
3. If only some parts of an aggregate are used by a function, pass only those parts.

The second and the third goal can and often are combined. If only an integer is read from a structure passed by reference, it makes sense to pass only that integer by value. Of course, pursuing these goals is subject to many limitations. Even when only individual parts of an aggregate passed by reference are used in a function, we cannot pass them separately (either by reference or by value) if the total size of parameters would grow too much. Of course, the original semantics of the program must be preserved, most importantly no new segmentation faults must be introduced to the caller.

3 Implementation

In order to regularize calls in such a manner, we have implemented an early interprocedural pass which has been committed to the pretty-ipa branch. As all early passes, it is not able to deal with indirectly recursive functions. The pass modifies functions in-place which means it can only transform functions if we can also identify and transform each and every one of their callers. Thus we have to exclude all functions which are exported from this compilation unit, those that have their address taken, virtual methods and so on. We have experimented with creating clones of non-local functions but rejected the idea because it would be difficult to limit the size increase relatively to the size of the whole unit and because there is currently no infrastructure in GCC for early passes to create clones and have them processed by subsequent early passes. A full-blown IPA pass capable of overcoming these limitations and a requirement for having this transformation working in an LTO environment, is an option for future work.

After the pass deems a function suitable for transformation according to the criteria above, it operates in the following stages:

1. The pass examines parameters which are aggregates or pointers and uses a candidate bitmap to mark those suitable for replacement by their parts or to a parameter passed by value according to their properties.

2. It scans the function body, examines all accesses to memory and whenever they involve any of the candidates, creates an access structure to mark the offset and size of the access. Moreover, if such an operand precludes us from breaking up any of the candidates (for example when the size or the offset cannot be determined or are not compile time constants), the candidate is removed from the bitmap. The analysis also disqualifies parameters if their addresses are passed to another function, stored in memory or is used as an operand (either to a pointer arithmetics statement or a PHI node).

3. The pass sorts all accesses for each parameter and searches for any overlaps. If there are any, the parameter is also disqualified. Otherwise, the pass finds a representative access for each combination of offset and size and creates a linked list out of these representatives. If there are any representatives of parameters which are passed by reference but which are not written to, the optimization walks the function again, trying to prove that no store through an alias or a side effect can modify the associated data and that associated parameters are always dereferenced when the function is run.

Then decisions are made as to what parameters are to be split into what components and this decision is represented in form of vector of struct ipa_parm_note (see section 5). Each structure describes one parameter of the function after the function is modified (and how it relates to an original parameter) but may also represent a decision to remove a parameter altogether.

4. Finally, the pass modifies the function and the call sites to reflect the decisions. It adjusts both the function declaration and its type and traverses its body again, replacing all references to the chosen parameters or their components to the newly created ones. Eventually it also alters all callers of the
function so that they pass parameters adhering to the new prototype.

A few aspects of the above scheme deserve a bit more thorough explanation. Before we decide to change a parameter passed by reference to one passed by value, we have to prove three conditions. First, we must not directly modify the parameter. Determining that is easy, we scan the gathered access structures which contain this information. Second, we must prove that the parameter is not modified through an alias before we read its value. This is a tough problem because interprocedural alias information is not available for the early passes (IPA-PTA has not been calculated yet) and therefore we resort to a simple method\(^1\) that works only for parameters that are read only in the first basic block provided that it has only one predecessor. If a parameter is read in it we store the number of that statement in the access structure. Then we traverse the whole function again, looking for statements which might indirectly modify a parameter. When we find one, we mark all parameters passed by reference as modified, unless they are only loaded in the first basic block and their position in that block guarantees it is not affected. In the future we hope to improve this detection by employing some type based alias analysis. Improving this detection is probably the key to making the whole transformation more useful.

Third, we need to prove that the parameter is actually dereferenced on all paths in the function because if it is to be converted to one passed by value, it might be necessary to dereference it in the caller. We do that by traversing all dominators of the exit basic block looking for such dereferences. Unfortunately, GCC dominance infrastructure is currently unable to provide the immediate dominator of the exit block and so this approach currently works only if it has only one predecessor and we resort to simply scanning the first basic block if it has more. This is very unfortunate because each loop is potentially infinite and we would need to add fake edges from all of them to the exit block. However, that would make it have more predecessors and so we resort to scanning only the first basic block also whenever we find a loop in the function. Making the dominance information work on the entry and exit blocks is thus also very important.

The decision whether a particular aggregate should be split into a number of its components is based on the size of the new parameters. If the original parameter was passed by value, we ensure that the sum of considered components is smaller than the original single parameter. If, on the other hand, the original parameter was passed by reference, we allow the new parameters (whether they are references or direct values) to be twice the size of a pointer.

4 Intraprocedural Scalar Replacement of Aggregates

The second step in the general scheme outlined in section 3 – scanning of a function when looking for accesses – can be efficiently reused by intraprocedural Scalar Replacement of Aggregates (SRA). Because I have been working on a new implementation of this pass too, I did make the two passes share this part. The new intraprocedural SRA was designed to be more simple and straightforward, use get_ref_base_and_extent to analyze aggregate references and be able to

\[\text{struct X}\
\text{struct Y}\
\text{union U}\
\text{struct C c}\
\text{struct Y y}\
\text{struct C f(double p)}\
\text{struct X x;}\
\text{struct Y y;}\
\text{struct C c;}\
\text{union U u;}\
\text{u.y.d = p;}\]

Figure 2: An overlap that precludes intraprocedural SRA.

\[^1\text{This part of the pass was written by Jan Hubicka.}\]
handle unions. It also operates in four stages (identifying candidates, scanning the function for accesses, access analysis, and modification), the other three are however substantially different. The first important distinction is that it only includes aggregates that do not need to live in memory in the candidates for replacement. Also note that even much of the scanning is common, each of the two passes has to perform some minor tasks the other one does not. For example, SRA does not need to scan PHI nodes whereas call regularization does not track assignments of aggregates to facilitate copy propagation.

The way collected accesses are analyzed is also very different. They are also sorted and representatives of each offset and size combination are identified too. On the other hand, intraprocedural SRA permits most overlaps of accesses, namely those when one accesses contains the whole area of another. Nevertheless, if, for any given candidate there is a partial overlap in between two accesses it is also disqualified. Two accesses partially overlap when they share a portion of the aggregate but each also spans outside the other one (see figure 2). Afterwards, the pass builds a set of trees from the access structures, in which the fragments of an aggregate associated with children are all within the part of the same aggregate which is represented by their parent. Figure 3 gives an example. Scalar leaves of this tree are then marked to be replaced by new scalar variables.

Finally, intraprocedural SRA also traverses the function in order to modify statements referencing the aggregates which are to be broken up into scalar pieces. This time, however, whenever it discovers a reference to a non-scalar access with scalar children, it has to deal with them too. If this happens in an assignment statement, the pass attempts to load scalar replacements on the left hand side from those on the right hand side when both sides have scalar replacements. When only one side has them, the transformation tries to load or store the replacements directly from the other side. If it is successful and the statement s no longer needed, it is deleted. The pass falls back to simply storing and loading replacements from their home aggregate and relying on the original statement to copy data if none of this is possible – this often happens when VIEW_CONVERT_EXPRs or unions are involved. In other than assignment statements, the data in replacements must simply be stored in the aggregate before it is read and must be loaded from it after it is written to (for example on the left hand side of a call statement).

5 Parameter notes infrastructure

At the moment there is no infrastructure in GCC for modifying formal parameters of functions and actual arguments of call sites. Yet, call regularization is not the first pass adjusting them, interprocedural constant

---

2A bit counter-intuitively, vectors and complex numbers are also considered scalars by SRA. A type is considered scalar if is_gimple_reg_type returns true for it.
propagation [7] deletes unused and constant parameters. Clearly, both passes would benefit from an infrastructure handling such manipulation, all the more so because they both should generate reasonable debug info for the affected functions. Moreover, as we envision more fully-interprocedural passes performing such modifications, often on the same virtual clone, a general mechanism to do this will become an essential part of the call graph.

Therefore I have written functions to adjust both formal and actual parameters according to the same specification which is a vector of struct ipa_parm_note defined in ipa-prop.h. Each such structure represents either a parameter of a function after the transformation or an intention to remove a preexisting parameter. The latter is necessary in order to have a place to hold debug information about the removed parameter which can be useful for example when the parameter is removed because it is known to be constant. The new parameters represented by such a note can either be intact copies of the original ones or specified fractions of an original one. Selected fields of the structure that are important for callers are briefly described in table 1. The infrastructure currently consists of a function to modify formal parameters of a function (ipa_modify_formal_parameters), a function to change actual arguments of a call statement (ipa_modify_call_arguments), and one to combine two vectors of notes into one (ipa_combine_notes).

### Table 1: Fields of the note structure.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_index</td>
<td>Index of the original parameter this structure relates to.</td>
</tr>
<tr>
<td>copy_param</td>
<td>True if the original parameter should be left intact.</td>
</tr>
<tr>
<td>remove_param</td>
<td>True if the original parameter is to be removed.</td>
</tr>
<tr>
<td>by_ref</td>
<td>True if the new parameter should be passed by reference.</td>
</tr>
<tr>
<td>type</td>
<td>Type of the new parameter, determines its size if passed by value.</td>
</tr>
<tr>
<td>offset</td>
<td>Offset of the new parameter within the original one.</td>
</tr>
<tr>
<td>nonlocal_value</td>
<td>Debug-info for a removed parameter, will be put to NONLOCALIZED_VARS.</td>
</tr>
</tbody>
</table>

### Table 2: Execution time of DLV [8] benchmarks without and with call regularization.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Without</th>
<th>With</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRATCOMP1-ALL</td>
<td>2.38</td>
<td>2.35</td>
</tr>
<tr>
<td>STRATCOMP-770.2-Q</td>
<td>0.34</td>
<td>0.34</td>
</tr>
<tr>
<td>2QBF1</td>
<td>15.72</td>
<td>15.16</td>
</tr>
<tr>
<td>PRIMEIMPL2</td>
<td>8.54</td>
<td>8.07</td>
</tr>
<tr>
<td>ANCESTOR</td>
<td>127.05</td>
<td>131.03</td>
</tr>
<tr>
<td>3COL-SIMPLEX1</td>
<td>3.83</td>
<td>3.78</td>
</tr>
<tr>
<td>3COL-LADDER</td>
<td>80.70</td>
<td>101.42</td>
</tr>
<tr>
<td>3COL-N-LADDER</td>
<td>1.55</td>
<td>1.52</td>
</tr>
<tr>
<td>3COL-RANDOM1</td>
<td>6.78</td>
<td>6.72</td>
</tr>
<tr>
<td>HP-RANDOM1</td>
<td>11.91</td>
<td>11.90</td>
</tr>
<tr>
<td>HAMCYCLE-FREE</td>
<td>1.50</td>
<td>1.61</td>
</tr>
<tr>
<td>DECOMP2</td>
<td>13.53</td>
<td>13.65</td>
</tr>
<tr>
<td>BW-P4-Esra-a</td>
<td>118.59</td>
<td>117.25</td>
</tr>
<tr>
<td>BW-P5-nopushbin</td>
<td>12.18</td>
<td>12.06</td>
</tr>
<tr>
<td>BW-P5-pushbin</td>
<td>8.77</td>
<td>8.81</td>
</tr>
<tr>
<td>BW-P5-nopushbin</td>
<td>2.33</td>
<td>2.27</td>
</tr>
<tr>
<td>3SAT-1</td>
<td>27.12</td>
<td>25.91</td>
</tr>
<tr>
<td>3SAT-1-CONSTRAINT</td>
<td>12.55</td>
<td>11.49</td>
</tr>
<tr>
<td>HANOI-Towers</td>
<td>4.30</td>
<td>4.40</td>
</tr>
<tr>
<td>RAMSEY</td>
<td>4.46</td>
<td>5.08</td>
</tr>
<tr>
<td>CRISTAL</td>
<td>8.73</td>
<td>9.80</td>
</tr>
<tr>
<td>HANOI-K</td>
<td>35.92</td>
<td>33.11</td>
</tr>
<tr>
<td>21-QUEENS</td>
<td>11.11</td>
<td>10.56</td>
</tr>
<tr>
<td>MSTDir[V=13,A=40]</td>
<td>13.07</td>
<td>12.35</td>
</tr>
<tr>
<td>MSTDdir[V=15,A=40]</td>
<td>13.10</td>
<td>12.41</td>
</tr>
<tr>
<td>MSTUndir[V=13,A=40]</td>
<td>8.60</td>
<td>7.90</td>
</tr>
<tr>
<td>MSTUndir[V=15,A=40]</td>
<td>130.57</td>
<td>123.39</td>
</tr>
<tr>
<td>TIMETABLING</td>
<td>7.28</td>
<td>7.69</td>
</tr>
</tbody>
</table>

### 6 Benchmarks

In order to assess the benefit brought about by call regularization, we have run a series of C++ benchmarks. However, because GCC uses inlining extensively on the benchmarks that we use, it tends to inline the functions call regularization modifies. Therefore we have deliberately turned down inlining limits when running the benchmarks. (3COL-LADDER benchmark had big variance in all runs for some reason and so the regression hopefully isn’t as big as presented here.)
### Table 3: Number of times individual transformations occurred when compiling Tramp3D [2], DLV [8] and Xpdf.

<table>
<thead>
<tr>
<th>Action</th>
<th>T3D</th>
<th>DLV</th>
<th>Xpdf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formal scalar by-ref. parameter converted to a by-value</td>
<td>285</td>
<td>779</td>
<td>2</td>
</tr>
<tr>
<td>An existing parameter split</td>
<td>1224</td>
<td>4107</td>
<td>858</td>
</tr>
<tr>
<td>New formal parameter replacement created</td>
<td>1787</td>
<td>6015</td>
<td>898</td>
</tr>
<tr>
<td>Formal parameter removed</td>
<td>3752</td>
<td>7464</td>
<td>5</td>
</tr>
<tr>
<td>New actual argument replacement created</td>
<td>8702</td>
<td>17994</td>
<td>4727</td>
</tr>
<tr>
<td>Actual argument removed</td>
<td>9771</td>
<td>12120</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 4: Effect of relaxing inlining limits on compile time of Tramp3D [2] at -O2 with early inlining turned off.

Figure 5: Execution time of Tramp3D [2] as inlining limits are tightened, with and without call regularization. Compiled at -O2 with early inlining turned off.

The effects it had on compile time and run time of Tramp3D are displayed in figures 4 and 5. In both these graphs the x-axis is factor of inlining limits, ranging from 10% to 100%, early inlining was switched off. The execution times reveal that when inlining is limited to a minimum, these transformations indeed have some small positive effect but are quickly overshadowed by inlining when given just a bit more space. The effect on the compile time is not significant. Execution times of DLV benchmarks is presented in table 3, only a few benchmarks improved slightly, but most of them did not change significantly.

The performance of the new intraprocedural SRA in terms of both the run time and compile time is on par with the old one. However, its main aim was simplification and handling of unions, so this is a good and
expected result.

7 Conclusion

This paper describes two new passes which have fairly different purpose but work in a similar fashion as they look for opportunities to break up aggregates. We have described their internal structure and the most difficult issues they need to solve. Above all, call regularization, when considering whether it can turn a parameter passed by reference into one passed by value, deals with potential aliasing and dereference legality in very simple ways and we believe improvements in these two areas may bring about more substantial benefits in the generated code. Because not only call regularization but also other passes need to modify function parameters and thus change their, declaration, types, bodies and call sites, we have developed and presented here an infrastructure based on parameter notes to handle these changes.

Finally, we have discussed various statistics and benchmark results to assess the usefulness of call regularization. We have discovered that at the moment all of the benefits are overshadowed by inlining, as inliner is likely to integrate the simple functions that call regularization is more likely to transform. On the other hand, we have discovered that removal of unused parameters alone is certainly a worthy goal for C++ scientific applications. Moreover, even the more complex transformations do take place on as many as 11% formal parameters. If we improve handling of the two issues stressed in the previous paragraph and extend the technique to a full IPA pass capable of operating within the LTO framework, it can deliver tangible benefits especially when compiling large programs.

Acknowledgments

I would especially like to thank Jan Hubicka for guiding me in my effort to design and implement these passes and Richard Günther for his help with many gimple issues.

References


Using Eclipse for Reverse, Multi-Process and Non-Stop Debugging with GDB

Marc Khouzam
Your affiliation
your-address@example.com

Abstract

The next release of GDB is full of major improvements aimed at providing the user with a richer debugging experience. Some of these features open the door to using GDB for debugging live telecom systems. Non-stop Multi-threaded debugging as well as Multi-process debugging are of particular interest in such scenarios. However, dealing with GDB when a thread is stopped but many others are running adds a level of complexity that makes it hard for a user to sort through the events of the debugging session. Multi-process debugging only adds to the amount of information that must be handled concurrently. A graphical front-end provides a much needed way of handling that information and presenting it to the user in a fashion that is clear and easy to visualize.

The next release of Eclipse, baptized Galileo, will provide DSF-GDB as part the C/C++ Development Tools project. DSF-GDB is a full-fledge integration of GDB into Eclipse which makes use of the latest features planned for GDB 7.0. Both GDB’s Non-stop Multi-threaded debugging as well as Multi-process debugging are used by DSF-GDB. Furthermore, DSF-GDB has already been enhanced to take advantage of the new Reverse Debugging feature that has just been added to GDB’s mainline. Finally, building on top of those enhancements, some new GDB and DSF-GDB features are already being planned such as Dynamic Tracepoints and GdbServer daemon.
Adding named address space support to the GCC compiler

Michael Meissner
IBM
meissner@linux.vnet.ibm.com

Abstract

The C language was designed for traditional machines with a single unified homogeneous address space where any pointer can point to any piece of memory, while some processors have multiple address spaces that the compiler needs to generate different code to access these name spaces. For example, the SPU processor has local memory and an extended address space to access host memory. Other ports might want to use the named address space support to add support for near and far pointers.

This paper will discuss the work done by Ben Elliston and myself to add the named address support from the ISO/IEC JTC1 SC22 WG14 N1169 technical report into the GCC compiler to support the SPU target. It will cover the basics of named address space support, the hooks needed to add named address space support, and the challenges to modify the compiler to support different named address spaces.

1 Introduction

The Cell Broadband Engine processor is a joint processor design between IBM, Sony Computer Entertainment, and Toshiba corporations. Sony uses it in its PlayStation 3 game console, IBM has Blade servers with Cell Broadband Engine processors and Toshiba has announced plans to incorporate it in television sets. The chip has some number (usually 8) Synergistic Processing Elements (SPEs) for doing the high performance computation and a a Power Processing Element (PPE) that acts as the host processor. The SPEs are the target of the GCC SPU target, and the PPE is a powerpc. Both the SPEs and PPE use the same memory system.

One reference to the cell design is at http://en.wikipedia.org/wiki/Cell_(microprocessor).

SPEs only have a small amount of local memory, and the PPE host processor can see each of the local memories for the SPE processors as well as its own normal memory.

The C standard stipulates that all memory objects are allocated in a single address space. The named address extension allows an implementation to have different address spaces where data can exist. The SPU target uses a single named address space, denoted by the __ea keyword to enable the SPU processor to access host memory that is not within its tiny default memory space.

2 The ISO technical report

The ISO/IEC JTC1 SC22 WG14 committee which standardizes the C programming language published a technical report (N1169) on April 4, 2006, which suggests several ways to add support to the C language for embedded environments. The reference number for the document is ISO/IEC TR 18037. The report can be found at: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1169.pdf

The report has several proposals. I will only talk about the named address space support in this paper.

- Fixed point arithmetic;
- Named address space support;
- Named register storage classes;
- Basic I/O hardware addressing.

The named address important features are:

- Address space type qualifiers: New address spaces are introduced with type qualifiers that bind like the volatile and const keywords.
• **Address space identifiers:** Address space qualifiers are normal keywords. An implementation may provide an implementation-defined set of *intrinsics* address spaces that are denoted with reserved keywords that begin with an underscore and an uppercase letter or two leading underscores.

• **Nonbuiltin address spaces:** An implementation may optionally support a means for adding new address space names. The current support for GCC does not provide a means to add nonbuiltin address spaces at this time.

• **No named address space functions:** Only data objects can be declared with named addresses.

• **No auto variables in named address spaces:** Named address spaces are limited to static/global data and to pointers. Auto variables cannot be declared with a named address space. You can however, have an auto variable that is a pointer to a named address space.

• **Generic address space:** Unless an item is declared with an explicit named address space, it is implicitly assumed to be in the generic address space.

• **Hierarchy of named address spaces:** All data objects are allocated into at least one address space. The implementation may have address spaces that are completely contained within another address spaces. Address spaces can be disjoint in that they do not share any memory locations. It might not be possible to convert from a pointer to one named address space to a pointer to another named address space.

• **Pointer sizes:** Pointers to named address spaces may be a different sizes, have different alignment requirements, or use a different encoding than the normal data pointers that do not point to a named address space. Pointer subtraction of pointers to named address spaces may similarly produce different sized results.

• **Pointers to void:** Pointers to named address spaces that point to the *void* type must have the same size and layout as pointers to *char* in the same named address space. You can convert from data pointers to a pointer to void and back again if all pointers point to the same named address space and get the same value back again.

• **Null Pointers:** Any two null pointers whose referenced address spaces overlap shall compare equal.

### 3 SPU named address support

This section covers the user visible portion of named address spaces.

#### 3.1 __ea keyword

The SPU compiler adds one new keyword, __ea, which says that something is in the external host memory instead of the local memory.

For instance:

- __ea char str[]: Declare *str* to be a variable in the host processor’s memory space.
- __ea char *p: Declare *p* to be a pointer that points to memory in the external address space. The storage for *p* is within the local address space.
- char *__ea q: Declare *q* to be a pointer to the local address space. The storage for *q* in the external address space.

#### 3.2 -mea32 and -mea64 switches

Control whether __ea pointers are 32-bits or 64-bits. The default is 32-bits.

#### 3.3 -mcache-size=

Control the size in kilobytes of the data cache that is used to hold host memory in the local address space.

On the SPU, libgcc contains a software cache implementation to the performance of variable accesses in the host address space. There are numerous cache configurations available for the programmer to select from the gcc command line.

#### 3.4 -matomic-updates

Automatically write back software data cache lines.
3.5 What happens under the covers

When you convert a local pointer to an __ea pointer, the compiler adds the offset of where the local memory is stored in the host address space, which currently is kept in the __ea_local_store variable.

Similarly, when you convert an _ea pointer to a local pointer, the compiler subtracts the offset of where the local memory is stored in the host address space.

If you dereference an __ea pointer, the compiler will call cache functions (__cache_fetch_dirty and __cache_fetch) to move the cache line from the host to the local memory and flush out the old cache line. If you are not compiling with -Os to save space, the compiler will generate inline code to see if the appropriate cache line is already loaded, and avoid doing the calls.

4 Hooks to add named address space support

Internally, address spaces are represented as a small integer in the range 0 to 15 with address space 0 being reserved for the generic address space. This limit is due to all of the C keywords are listed in the enum rid enumeration in c-common.h, and it can be increased if there is need for more named address spaces.

The named address spaces are kept in the type addr_space_t. Currently this type is an unsigned char, but it can be increased if needed. Note, the address space is kept in the tree type structure and the rtl mem_attrs structure used for all memory attributes.

- TARGET_ADDR_SPACE_KEYWORDS: This is a macro to register the named address space keywords, and map the keyword into a named address space identifier.
- TARGET_ADDR_SPACE_POINTER_MODE: This hook returns the appropriate machine_mode enumeration to hold the pointer type. For example, on the SPU, this returns DImode if -mea64 is used, and SImode otherwise.
- TARGET_ADDR_SPACE_MINUS_TYPE: This hook returns the appropriate tree type for an integer mode to hold pointer subtraction between two pointers to different named address spaces. On the SPU, it normally returns a 32-bit signed type, but it will return a 64-bit signed type for __ea pointers when -mea64 is used.
- TARGET_ADDR_SPACE_NAME This hook maps a named address space identifier into a string that the compiler can use in diagnostics for messages about incompatible named address spaces.
- TARGET_ADDR_SPACE_LEGITIMATE_ADDRESS_P: The named address version of the LEGITIMATE_ADDRESS_P hook that validates a pointer to a named address space.
- TARGET_ADDR_SPACE_LEGITIMIZE_ADDRESS: The named address space version of the LEGITIMIZE_ADDRESS hook that given an invalid pointer, tries to fix up the RTL to make a valid address.
- TARGET_SUBSET_P: Return true if one named address space is a subset of another.
- TARGET_ADDR_SPACE_CAN_CONVERT_P: Return true if you can convert from one pointer to a named address space to a pointer to a different named address.
- TARGET_ADDR_SPACE_CONVERT: Produce the RTL to convert a pointer to a named address space into a pointer that points to a different named address space.
- TARGET_ADDR_SPACE_SECTION_NAME: Return the section name to use to put static and global variables that are in a named address space.
- TARGET_ADDR_SPACE_STATIC_INIT_OK_P: Return whether a particular initialization is allowed for a static or global variable in a named address space. The SPU does not allow initializations that require relocation to go into the __ea named address space.

5 Experiences in adding named address support

The named address extension is one of these extensions that touches all of the compiler and consequently has a number of diversions.
5.1 Front end vs. back end engineers

One of the problems during development is the work was done by backend folk that were touching front end parts of the compiler. Over the year when first Ben and later I were doing the work, we would put in changes that didn’t quite match the requirements on the C front end. In most of the time, when we went back to look at the issue, we were wrong and the front end developers were right. As I alude to in my other paper (Hackers are from Mars, Corporations are from Venus) in this conference, it is important to actually listen to what the reviewers are saying, and not to be as hardheaded about having to have things done the way you originally implemented them. *Mea Culpa.*

5.2 Corporate reassignment

Another problem that we faced in doing the development was that of corporate reassignment. Originally Ben had done the initial work, and then was moved off of the project, and I had to take it over, and come to speed on what the changes that Ben had done. Now, I too have moved on, leaving the code to Ulrich.

5.3 Pointers are not integers

The biggest problem in adding the named address space support is that the GNU compiler was originally written to think of pointers as synonymous with integers, instead of having a pointer abstraction. This means if you want to add things like pointers to named address spaces have to find all of the places that inadvertently confuse pointers and integers and work around the problem.

5.3.1 ISO C standard

In terms of the ISO C standard, there are a few places where the standard mandates behavior that requires pointers to obey some rules:

- **Void * and Char * must have same encoding:** This was needed because originally there was no `void * pointer, just char *`, and the ANSI/ISO committee wanted to make some functions like `malloc` that previously returned `char *` now would return `void *`.

- **Structure and union pointers must have the same encoding:** This is required to allow declaring forward reference pointers without declaring the structure or union body.

- **Void * must be able to point to any data object:** Having a generic pointer that can point to anything is a useful concept. It does break down somewhat with the ideas of named address spaces. Note, the standard does not require that `void *` be able to point to functions.

5.3.2 Pointers abstraction problems

When I first started working on GCC in the fall of 1989, I thought not having an abstraction for pointers was one of a few things that I thought the design was completely wrong.

There were and are a lot of things that I might wish things had been done differently, but only a few places that I felt were fundamentally flawed. If you are curious, some of the other things that I thought were bad abstractions are:

- **CC0:** Having a hidden register `CC0` for the condition code, and having the dependence between the insn that sets `CC0` and the jump or `setcc` instruction that uses it being implicit and having to keep the compare and jump next to each other throughout the compilation instead of explicit;

- **Trampolines:** The concept of using trampolines to call to lexically scoped functions is just wrong (see below).

- **Structure and union pointers must have the same encoding:** This is required to allow declaring forward reference pointers without declaring the structure or union body.

- **Void * must be able to point to any data object:** Having a generic pointer that can point to anything is a useful concept. It does break down somewhat with the ideas of named address spaces. Note, the standard does not require that `void *` be able to point to functions.

The compiler, is a lot better about abstractions for pointers now, than it was twenty years ago. The front and middle ends of the compiler carry the full type around, so you have fewer places where you can lose the information. The RTL backend has the `mem_attr` structure to collect the memory abstraction, but every so often you find places that do not use the sanctioned interfaces for updating the address. Unfortunately, the only way to discover these places is by running code through the compiler, and finding where stuff breaks.

---

*Mea Culpa*
5.3.3 Pointers are not integers problems

Some of the things I have run into over the years where this notion that all pointers look and smell the same is a problem for real machines (perhaps not mainstream machines, but machines that people did design):

- **Machines with different pointer types**: When I first encountered GCC, I was working for a company (*Data General*) that had a machine that had different representations for pointer to character and pointer to word. On the Data General machine, the pointers were the same size, but the character pointer was shifted left one bit, and the bottom bit indicated which byte within a 16-bit word was referenced. The word pointer originally had an indirection bit that was lost in character pointers. Other machines of the era weren’t as lucky in that they needed an extra word to store the byte location, separate from the word pointer. In the proprietary C compiler front end that I wrote for the Data General machine, it exposed how cavalier C programmers were in passing pointers to functions that were one type and dereferencing the pointer as an other type.

- **Near and far pointers**: Near and far pointers in the 32-bit x86 environment would fit within the named address space support, but are not representable within the current GCC.

- **Word accesses on alpha and spu**: While the pointer format on the *alpha* and *spu* uses a universal format, the backend has to synthesize accesses to smaller fields.

- **Trampolines**: As I mentioned above, the ISO C standard does consider function pointers to be different than pointers to data. By having to fit function pointers into the normal data pointer straight-jacket GCC does calls to lexically scoped functions by creating an area on the stack to hold the instructions that load up the register holding the static chain, and then jumping to the real function.

I have run into the following problems over the years with trampolines:

- **Flushing the data/instruction caches**: To implement a trampoline correctly, you need to flush out the data cache after creating the trampoline, and invalidate the lines in the instruction cache. The original *MIPS* chips had no instructions that a non-privileged user could use to properly flush out the data cache and invalidate the instruction cache, and you had to issue a system call to flush the cache, which has a number of disadvantages. On some machines, the instruction sequence to flush the cache properly varies, depending on which machine you are executing on within a family.

- **Machines where the text area is not readable**: On some embedded processors I have worked on over the years, such as the *Mitsubishi D10V*, the area where the code is located is not readable by the program. Several of the *PDP-11* computers also had the notion of having the instructions in a different address space than the data, and the same address could represent a function address and a piece of data, it was only using it in a function or data context that made it unique. If the text area is not visible as a data item, it is impossible to make a trampoline on the fly to call a lexically scoped function through a pointer.

- **Operating systems that don’t enable executing code on the stack**: Some operating systems don’t make the stack executable. Similarly to the machines, where the text is not in the data address space, if you can’t write to a piece of code, and easily make it executable, you can’t make a trampoline. Sometimes you can issue a system call to make the stack executable and sometimes you can’t. Modern versions of the *Linux* operating system do not make the stack executable, unless the compiler marks in the object file that this program uses trampolines. Many years ago, I attempted to make trampolines more abstract and not be required to reside on the stack, but I was unable to find all of the hidden assumptions for everything having to reside on the stack, and eventually gave up.

- **Support for -m32 and -m64**: Several of the back-ends like the *x86* and *powerpc* now support 32 and 64-bit execution environments. This means you
have to clone the insns that deal with pointers to handle 32 and 64-bit pointers. If pointers had been a separate machine type instead of just being integers this duplication could have been avoided. Of course on the other hand, it would mean you would have to duplicate some of the insns like `addsi3` to handle pointers.

5.4 Changes to the C front end

The following changes were made to the C front end:

- **Parser:** The current design has the backend statically registering the named address space keywords via the `TARGET_ADDR_SPACE_KEYWORDS` macro. Up to 15 keywords can be registered at present. This was done to fit into the current framework of initializing keywords in the global `c_common_reswords` array and to have a 1-to-1 correspondence between the `RID` enumeration and the keyword.

Originally this was done by having a target hook callback for every identifier that wasn’t a keyword to see if it was a named address keyword, but it was felt that this would slow down the compiler too much.

One possible implementation for the future would be to have the backend dynamically register keywords via a target hook. While it is nice to be general, I don’t think implementations really want to register different keywords based on the switches used. In particular, it will probably cause implementation problems if a backend wants to change what keywords are used based on the `target` attribute or pragma.

- **Declarations and Type checking:** Most of the C front end specific changes went into adding support for declarations with named address spaces, and for type checking between similar types. As I mentioned earlier, it was hard for us to get the nuances correct with regard to where the named address qualifiers are allowed and where they weren’t.

- **Initializations:** We needed to add code in handling initializations to add support for `TARGET_ADDR_SPACE_STATIC_INIT_OK_P` to allow an initialization of an item residing in a named address space or a pointer to a named address space or not. For the SPU, we do not allow initializations that involve a relocation. The reason is under the SPU, the `__ea` declarations are declared in the module and put into a section, and then these are copied to their final location when the program starts. The address is determined at runtime, and we do not maintain a list of pointers that need relocating once the copy is done.

5.5 Changes to the tree infrastructure

The following changes were made to the parts of the compiler dealing with trees:

- **Basic infrastructure:** Changes were made to keep track of the named address space information. Named address spaces are encoded as a multi-bit field with the other qualifiers. Some places where the compiler keeps or removes qualifiers it makes sense to deal with the named address space qualifier, and in other places it makes sense to remove the qualifier.

For example, in the places where `and` and `or` operations are done on type qualifiers, you have to separate out the named address space, do the logical operation on the other qualifiers, and then handle the named address space.

- **Pointer sizes:** Pointer conversion, pointer subtraction, storage layout, and a few builtins needed to be converted to deal with the fact that pointers to named address spaces might be a different size than the standard pointer.

- **Dwarf2 support:** The `dwarf2` module was modified to emit the `DW_AT_address_class` die to the debug information for the named address space that a variable is in if it isn’t in the generic address space, and what named address space it points to if it is a pointer to a non-generic address space.

5.6 Changes to the RTL infrastructure

The following changes were made to the parts of the compiler dealing with trees:

- **Memory attributes:** The `mem_attrs` structure was modified to add a field for the named address
space that the memory reference points. All of the functions that modify and inspect the `mem_attrs` structure needed to be modified to deal with the additional field. In writing this paper, it occurs to me that it may be useful to track the attributes in the `mem_attrs` structure to allow back ends to do similar things to named address spaces for machine specific purposes.

- **Target hooks**: The target hooks that deal with verifying and converting unsupported addresses into supported addresses needed to be cloned to have named address space versions. In an ideal world, it would be better to combine these two hooks and have a pointer abstraction that takes a structure for the optional attributes that could be added to in the future as new things come up.

During the course of development, Paolo Bonzini liked the idea of doing the memory addresses as a single target hook, and changed all of the back ends to move `GO_IF_LEGITIMATE_ADDRESS` from an old-style macro to a target hook, and remove the macro. After he made the changes, I then went back to the named address space branch, and changed the hooks to combine the strict and non-strict address hooks into a single hook.

- **Pointer usage**: All of the places that used `Pmode` as the default pointer machine mode, had to be inspected and possibly modified to use the `pointer_mode` hook for the machine mode if the pointer pointed to a non-generic named address space.

- **Forming and modifying addresses**: Functions like `change_address` needed to be modified and/or cloned to deal with pointers to different named address spaces.

### 5.7 Changes to the SPU backend

The following changes were made to the SPU backend:

- **Target Hooks**: All of the target hooks mentioned above need to be implemented. We added rtl support for converting from `__ea` pointers to generic pointers and vice versa.

- **Cache controller**: We added the cache controller to `libgcc` that the compiler calls to move data items to and from the host memory.

- **Macros**: If `-mea64` was used, signaling host addresses are 64-bits, we define `__EA64`, otherwise we define `__EA32`.

### 6 Future

Here are some thoughts on how things that should be done to the named address support in the future.

#### 6.1 Get the changes into the mainline

It would be nice to get these patches into the mainline. I suspect there are some other embedded ports that could use the abstraction of named address support for things like `near` and `far` pointers.

#### 6.2 Add C++ support

Right now, only the `C` language has support for named address spaces. It would be useful to dive in and add the `C++` support as well. Just as this support goes into the nuances of the `C` front end, I imagine we would need somebody who knows the `C++` language quite well to add the necessary support.

#### 6.3 Dynamic keywords

The technical report that talks about named address spaces (N1169) mentions that implementations might provide a way to add new named address spaces. I’m not sure whether there is enough need to do this, but if it is needed, we would have to rethink how to register new keywords dynamically, and more importantly how to unregister the keywords. At the very least, it may be useful to change to use a target hook to provide the keywords instead of providing them in a static table.

#### 6.4 Add proper pointer abstraction

Rather than cloning the various pointer abstractions for generic vs. named address space, it is probably useful to sit down and design a proper abstraction layer that would allow the back ends to add new features as they come up without modifying the rest of the compiler.
6.5 Reverse endian memory

One of the things that comes up from time to time in the embedded world is adding support for explicit little or big endian memory declarations. Most of the work that is needed to be done for reverse endian loads and stores is essentially the same that is needed to be done for named address spaces. I could imagine a back end could have have a reverse endian named address space that would allow it to generate the appropriate swaps on loads and stores.

6.6 Expand named address pointers at the tree level

Right now, the named address space support is generated at the RTL level. For things like the calls that the SPU generates for the cache, it may be useful to have a hook to expand the address at the tree level, perhaps writing them as builtin functions.
Hackers are from Mars and Corporations are from Venus

Michael Meissner  
IBM  
meissner@linux.vnet.ibm.com  

David Edelsohn  
IBM  
edelsohn@us.ibm.com

Abstract

Hackers and corporations do not always talk the same language and often times have conflicting goals. It can be difficult for corporations to work in the free software space to achieve its goals and just as difficult for the hackers to interact with the corporations.

In this paper, we will cover various problems that often arise due to these culture clashes, and suggest strategies for overcoming the problems.

1 Introduction

Over the last 20 years, we have seen the same dance over and over, of corporations or individuals trying to get changes that they want into the Gnu Compiler Collection (GCC), the binary utilities (assembler, linker) or debugger (gdb), and getting frustrated at the process. At different times and places, we have been on both sides of the table, as an employee of a corporation trying to get our changes in, and as a maintainer for the Free Software Foundation trying to improve the toolset and telling people to change their code. It is kind of tiring to see the same things happen over and over again.

The points we offer in this paper are not magic bullets guaranteed to get your code in without any problems, but suggestions as to ways to make the process less painful. If you look up our posting history with patches, you will see that even after all of this time, we often times have complaints about the patches, and have to go back and rewrite patches to make things acceptable.

It is likely that many of the people attending the talk at the GCC summit probably have already been through the hurdle of getting their code into the toolchain. But everybody has to start sometime, and this paper hopefully will talk to the people contemplating adding their changes.

We also hope the paper is helpful to explain to management of companies contemplating contributing what is involved. It is better to know what the road is going to be like ahead of time, rather than rushing in and finding things are different than you expect. For example, over the years, we have had several different managers, and we have had to explain the ins and outs of contributing to the FSF each time.

1.1 Disclaimer

Generally disclaimers are implicit, but this talk is not a standard technical talk, and it is important to actually state the disclaimer in print. The views expressed in this paper and talk are that of the authors, and do not necessarily reflect those of our current or previous employers. We are hackers, and not lawyers, so seek your own lawyer for interpretation of the finer points of the GNU public license (GPL).

1.2 What’s in a name?

In this talk, we call the people who work on free software projects “hackers”, which is in the context of the free software community means developers. It does not mean people who try to break into systems, plant viruses, etc.

We will also use the term toolchain to refer collectively to all of the elements of compilation owned by the Free Software Foundation (gcc, assembler, linker, disassembler, debugger).

1.3 Stereotypes

One of the reasons we are writing this paper is both sides tend to have stereotypes of the other. It can be easy to fall into the trap where corporations view hackers as
anti-social technology nerds, and hackers view corporations as soulless entities trying to stamp out individuality. The problem with stereotypes is they can get in the way of actually getting what you want done if you feel the other side is morally objectionable. At the end of the day, we are all people, and we view the world from many different perspectives.

1.4 People vs. organizations

- One of the stumbling blocks that the two cultures run into is people vs. organizations. In general, due to their nature, corporations, tend to view things in terms of organizations, while within the free software world things are organized by individuals. This can be a stumbling block, because there is no management to go to, and instead the corporation has to deal with individuals, all of whom come to the process with different motivations and skill sets.

- For a corporation, this means the best strategy is having people who interact with the free software community on an on-going basis, since the free software community is people based. If you rotate people into a slot to deal with the community and then rotate them out again, there isn’t much continuity, and often times the new people have a hard time being heard. Most developers are more likely to respond to people that they’ve had previous dealings with. Now, this can either be working with developers who are on the inside track, or it can mean hiring your own people to interact with the developer community.

- Similarly, most developer communities do not defer to “experts”. Someone may have designed and implemented an important feature in a company product or have an important title in the company or be an academic expert in a particular topic, but that does not mean they understand the community’s codebase or priorities. Deploying a company “expert” to tell the community what it should do is unlikely to be effective. If the person participates in the community and demonstrates that his implementation is best, the community will incorporate the contribution.

- Another difference in approach is hierarchy. Most free software development is very distributed with leaders and gatekeepers who have developed expertise in various areas. Large corporations, on the other hand, have centralized authorities “architects” and “design councils” who create the design and plan and give out marching orders. The leader or leaders of a free software project rarely can direct the community to develop or accept a feature proposed by a corporation. One must convince the gatekeeper responsible for that domain.

- A lot of the issues in getting patches approved are a variant of he said/she said type issues, where there is no true communication going on. It may help to step back, and more clearly explain what the problem is that you are trying to solve, rather than assume the other party knows everything you do. GCC is a very complex beast, and it runs on a lot of environments, and most people do not know the ins and outs of every single platform.

- In the past there was a phrase called “Egoless programming”. In general there is no such thing, but it is important to realize in dealing with another person, that either your ego or theirs might be getting in the way. They may be right and you might be wrong (of course it can also be you are right and they are wrong, or some mixture in between). A lot of the fights in computer developer circles is due to ego (and this applies both within the free software community as well as within development groups inside of corporations, but you don’t always see the battles for non free software groups).

- Even if you think the other side is wrong, it generally works better if you keep things at the purely factual level, and keep your own ego in check.

- Sometimes however, you can get in such a rut that no forward progress can be made. It may make sense to replace the employee with a different person who has no history, can get things started again. Often times being different means that person doesn’t have as much stake involved in the code as it is, and might be willing to make changes.

- In photo sharing forums, you often read complaints from people who are new and put up a picture for comment that why does Jane Smith always get comments when she posts, but nobody looks at my picture, which is similar to the I posted a patch and
nobody looked at it comments that are heard every so often. If you look at the posting history, the person who gets lots of comments, tends to be the person that spends more time interacting with the other people, and people are commenting because they know the poster. The same thing tends to happen in terms of patches.

- Another analogy to the photo forums is if you post a picture with a subject line that doesn’t describe the photo many viewers will skip over the picture. In the same fashion, if your subject line for a patch is something generic like just the bugzilla number, or says crash, many reviewers who are short on time will skip over it.

1.5 What hat is a developer wearing

Developers in the free software community often times wear many different hats. Sometimes they are speaking for themselves, sometimes they are speaking in the context of the maintainer of a certain part of the compiler, and sometimes if they are employed to work on free software, they might be speaking for their employer. People new to the process should try to figure out where the other person is coming from and what hat they are wearing.

Similarly if you are wearing different hats, you should indicate which hat you are wearing if it matters. This can be important, especially if you post email from a corporate email address, and most posters would assume you are speaking in behalf of the corporation.

1.6 Motivation

One thing that seems hard for companies to come to grasp with is motivation. In general, corporations tend to deal with motivations that are purely monetary based (I will pay you this amount of money for these services or goods in exchange). In the free software world, people will work on free software for other motivations in addition to monetary, such as wanting to optimize their particular application, working on a cool project, or to run on their particular piece of gear.

A lot of times, corporations do deal with this in terms of managing employees where the good manager tries to match the jobs to be done to the people being managed, but they don’t do this with external entities. Spending some time to understand where the other people are coming from and what problems they are trying to solve, might be able to minimize the differences of opinion.

Another thing that can be helpful is to realize that not all free software developers are the same. We each have different motivations and motives, and one size fits all approaches often times backfire.

1.7 Free Software Foundation

The Free Software Foundation, it was established on what it calls the four freedoms http://www.gnu.org/philosophy/free-sw.html:

- The freedom to run the program, for any purpose;
- The freedom to study how the program works, and adapt it to your needs;
- The freedom to redistribute copies so you can help your neighbor; and
- The freedom to improve the program, and release your improvements to the public.

The upshot is if you intend to modify the code owned by the Free Software Foundation and share it with other people, such as the compiler or binary utilities, you need to follow the rules that they have established. You are free to write your own compiler and binary utilities, just don’t use the code that is part of the toolchain in ways that violate the license agreements.

You are allowed to do your own modifications to the toolchain without restriction as long as you don’t distribute it in either source or binary form. The restrictions that the FSF puts on its software are meant so that if you modify the toolchain and distribute it, that you need to distribute it in a fashion that allows other people to be able to modify your changes and places no restrictions on their downstream use of the toolchain.

Sometimes you run into people with the idea that since GCC is free software that they can do whatever they want with it. It is not public domain software, it is under copyright. The owner of the copyright (FSF) has decided to place some restrictions on the software so that it furthers the goals that it was founded on.
1.8 Why go through this trouble?

As painful as getting through all of the hoops to contribute code is, there are good reasons for companies to contribute to the effort:

- **Customer requirement:** Some users require using GCC over other compilers, and if there is no support for a particular hardware setup, these users will go elsewhere. The cost of doing hardware enablement needs to be balanced against the possibility of lost sales.

- **Time to market:** A hardware company has prototype hardware available years or months before it is available for sale to the general public. If you wait for somebody to add the feature on their own, it may take years or decades for other people to write the support. The hardware may even be obsolete before somebody else adds the support.

- **Optimization and tuning for hardware vendors:** In general new computers that are in the same family as previous computers have somewhat different tuning parameters, and the hardware company has the best insight as to the best code generation strategy to use.

- **Optimization for applications:** Even if a company does not sell hardware, it may have a core application that it wishes to run as fast as possible, and paying for people to add optimizations that are not being done can give it a competitive advantage.

- **Board evaluation:** Embedded companies often times tailor GCC to support their evaluation boards with GCC, and put it out for free to convince people to buy their chips.

- **Linux and BSD:** GCC is the compiler of choice for Linux and BSD operating systems, and if a hardware vendor wants to sell into that market, they need to provide support for their hardware if there are appreciable differences in the hardware.

- **Giving users options:** No user likes being locked into an environment with no chance of switching. Even for companies that offer their own compilers, supporting GCC as a second compiler, gives the user a choice.

- **Bug fixing:** Even if a user never needs it, having the source available to GCC means that in a critical event, a user could patch the compiler on his own to work around a critical bug.

More information can be found at: http://gcc.gnu.org/contribute/why.html.

2 Legal stuff

One of the issues that corporations and individuals run into is the various legal issues in contributing to the Free Software Foundation. As we said above, we are not lawyers, and anybody should consult with their own legal experts on the precise meaning of the various licenses. These thoughts are things we’ve observed over the years, but it is not intended to be an in depth exploration of the Free Software Foundation licenses.

2.1 GPL, its not just a good idea, it’s the law

Every so often, companies run afoul of the GPL (Gnu public license) or LGPL (Lessor Gnu public license) that the toolchain and libraries are distributed under. Sometimes companies are intentionally violating the license, but a lot of times things happen because of invalid assumptions. The Free Software Foundation does protect its copyright of the toolchain sources, including using the law to enforce the requirements.

- If you release a set of toolchain binaries, you need to release the source that goes with it. Usually the best strategy for a corporation is to release both the source and the binary at the same time, and on the same media.

- If you release a set of toolchain binaries but not the source at the same time, you need to be able to release the sources for at least three years after you released the binaries, and longer if you are offering spare parts or customer service for hardware covered by the binary release. In practice this can be hard to do as a company has to guarantee that it will still provide the source, even if it was bought by another company. It is much easier to meet the requirements if you release the source at the same time as the binaries.
• You can distribute changes to the toolchain independently without trying to get the changes contributed back to the Free Software Foundation, but the changes must follow the various licenses. Typically, it is better to get the changes merged to the official repository, rather than maintaining your own patchset, but it is an option.

2.2 Contributing to the GNU compiler collection

The main rules for contributing to GCC are noted at:

• http://gcc.gnu.org/contribute.html
• http://www.gnu.org/prep/maintain/maintain.html

In terms of companies contributing to GCC, some things to keep in mind are:

• In a work for hire, a company can decide that the developer assigns rights to the company, and then the company donates the code to the Free Software Foundation, or the company can disclaim the rights to the software, and have the individual developer contribute the changes. We have worked in both situations, and generally the first option is the best.

• Unless you know you are going to be doing only one donation, it is better to setup the assignment for all future changes.

• In either case, somebody who has signing authority for the corporation must authorize the assignment. A lot of times, this can be a crucial step, in that the people doing the work need to present a business case why this assignment is needed.

• Sometimes the legal staff needs to be educated as to why doing this is a good idea, and that you aren’t going off on a wild goose chase.

• Until the paperwork is signed, you can’t get your changes into the Free Software Foundation repository. It happens frequently that work has started on a port without the developers having the ability to get the patches committed.

2.3 GPL v3 vs. GPL v2

In November 2007, the Free Software Foundation updated its main license (GPL) to version 3. This paper is not the place to go into the various differences between version 2 and version 3 of the GPL. Here is the site that discusses the license: http://gplv3.fsf.org/.

Some corporations have decided that they cannot adhere to GPLv3, and will stay at GPLv2. In the context of the toolchain, all active code development is now done at GPLv3. If a company’s legal staff feels that they cannot comply with the GPLv3, then they cannot contribute to the active development of the toolchain. This also means it limits the pool of developers who are willing to work only on older toolchains and not be able to contribute their work back to the Free Software Foundation.

2.4 Disclosure of future machines

One of the things that comes from time to time is when a corporation discloses new hardware changes. Because the toolchain source is freely available, developers cannot put support for new hardware into the Free Software repository that are enabled with a new switch until some of the details of the new hardware are announced. Typically this decision must be made by senior management of the company because this is seen as giving the competition advance warning.

This usually means that developers must work with their management, so that the announcement is made early enough that the changes can be put into the development sources. Often times, this pre-announcement is in the form of an update ISA (instruction set architecture) that describes new instructions that the compiler would generate, but not a lot of other details for the chip (cache size, processor speed, cycle speed, number of cores and processors, etc.).

3 Doing the work

Once you have cleared the legal hurdles, then you can start contemplating making changes to the toolchain. It out of the scope of this paper to discuss all of the issues in modifying the compiler, etc. Instead we will leave it as an exercise to the reader, and assume that you have code that you want to contribute back to the Free Software Foundation.
3.1 Release schedules

If you are making a change to work on certain hardware, you need to keep abreast of the state of the release cycle (http://gcc.gnu.org/develop.html). In general, there are several stages that the code goes through for GCC (the binary utilities and debugger have different cycles):

- Stage 1 – large features can be added
- Stage 2 – smaller features can be added
- Stage 3 – bugfixes only

GCC is starting to evolve towards a Stage 1/2, Stage 3, Stage 4 development model, diminishing the distinction between large and small features, while focusing more on bug fixes before (Stage 3) and after (Stage 4) the next release is branched but before it is released.

If you are adding support for new hardware, you should be aiming to contribute it in Stage 1, and if you miss the stage, you will need to wait until the next release.

One thing that comes up from time to time is the requirement that supporting new hardware is considered a new feature and not a bug fix. This means it may be a year or so before you can add support to the official tree. Common strategies for dealing with this are providing an advanced toolchain either directly or through a third body, like a university, or getting the Linux/bsd distributions to support your machine.

3.2 Keeping your code up to date

- It is often tempting to stay with a given version of the compiler while doing development so that you minimize the impact that external changes made by other developers. However, before you submit the changes, you will need to merge up to the current version of the sources. In the long run, it is less painful if you do the merges every so often, and fix the problems incrementally, rather than trying to do a big bang approach.

- One way to do this is to maintain two branches of checked out code. On branch is the mainline with no changes, and the other is the development branch. There are various scripts out there that help you do these merges automatically, such as a script called svnmerge (http://www.orcaware.com/svn/wiki/Svnmerge.py) that uses subversion to manage the merge step. When you are satisfied with the changes on the development branch, check them all into the branch. Update the main-line branch, and do a svnmerge step on development branch, and bootstrap/test both branches. If there are merge conflicts or errors in the test, commit the changes to the development branch and repeat the cycle.

- Most times, everything works automatically, but there are times when you will need to modify the code to take into account the changes to the main-line. It is a lot easier, when the amount of changes are smaller, than say two or three months worth of changes.

- If you are working on machine ports that have not been disclosed yet, you could maintain a subversion repository on a machine within your company. The main branch of the repository would be pure mirror of the mainline branch, and you would maintain a branch off of that to do development. When you want to do a merge, you would check out the branch from the Free Software development machine and then check the sources into your local repository. This way you can update and merge with the mainline at any time, while still maintaining the work you are working on.

- It is good practice to keep a private ChangeLog of the changes on the development branch, which keeps a running commentary of what you have done. You probably want to use a file named something other than ChangeLog for this private branch, so that the normal ChangeLog file is not touched.

3.3 Backports

Generally, compiler developers are one or more releases ahead of the users. Sometimes users will not consider moving to new versions of the compiler because they do not want to deal with the changes that the rest of the compiler will introduce (new optimizations, changes to language semantics, etc.). This means that if you are providing compiler support for a new piece of hardware, you probably need to plan to add this support to older versions of the compiler. In general, it will not be just a drop in change, but you may need to rewrite parts by
hand, and hand merge these changes in. The further back you go, the more painful this becomes.

For instance, as we write this paper, we have a backport of power7 changes added to branch based on a 4.3 GCC, and it has some subtle failures that need to be identified and fixes.

3.4 Private features

Some companies have created features in their version of the toolchain that is never merged back to the mainline. Sometimes these features are rejected from going in, sometimes they are done in a different fashion, and sometimes the features are never proposed for inclusion into the mainline. In any case, having private features can add significantly to the cost of maintaining the port, and it means your users are locked into using the alternative toolchain. It is the best practice for all code to be pushed to the mainline to reduce the cost of maintaining parallel toolchains. Sometimes this is needed to run an existing code base, but you should strive to make it rare.

Using private features often times locks you to a particular revision of the compiler that those features were developed on, particularly if the features were developed by a third party, and not with in-house staff who can move the code forward. Over time, it becomes harder to find people who are willing to keep this old code up and running, and supporting this code becomes more of a financial drain.

3.5 OS distributions

- Most users of GCC do not build the compiler themselves, nor do they necessarily download the compiler from a support site. Instead they usually rely on their OS distribution (Linux, BSD, cygwin, etc.) to provide a compiler. Some distributions see themselves as providing cutting edge tools, while others see themselves as selling stability, and update their compiler toolchain to new revisions only when the distribution goes to a new major version number.

- This means that it might take the average user years before they get to the compiler being developed right now (4.5 is being developed at the time this paper was written), factoring in users not upgrading their system and OS distributors only changing compiler base revisions once every 1-2 years.

- Companies that offer new hardware more frequently than that, need to have plans to get the distributions provide support for the new hardware in either their main compiler version or an advanced compiler version. Usually this is one or two versions back, which means backporting the changes from the development branches to vendor branches. Each OS distributor uses its own branch, which multiplies the number of branches that need to be tested.

- OS distributions usually require that the changes be merged upstream to the mainline development branch before accepting a backport. This can be problematical when the development tree is closed to new features.

- Usually what happens is a series of negotiations with each of the OS distributors of what patches to put in their next release. This multiplies the problems facing corporations with new hardware, since it means several different groups to make the case for the changes, and you need to spend a lot of time maintaining various trees.

- One thing that is a problem is often times hardware companies limit their involvement to just the top distributions. For example, in the Linux world for business computers, this would be Red Hat and SUSE. Now for things like mainframes this likely makes sense, since you don’t just go down to Best Buy and buy a series z mainframe, but for the x86 desktop/laptop world, it may be useful to also talk to the other distributions like Mandrivia, Ubuntu, etc. It becomes a job to balance the cost of dealing with multiple distributions vs. the payback in terms of not alienating potential customers.

3.6 Making it general

- One of the things that developers working for hardware companies tend to not think about is making the code that they are working on general enough for other platforms to use. If you are only modifying the machine specific files and not adding new machine dependent passes, this may not apply. If you are adding new optimizations in the machine independent parts of the compiler, you should strive to make your solution general enough for other ports to use.
• Usually this means soliciting feedback in the mailing lists or on IRC for other developers, and see if they have similar needs, and then working with them to come to a common goal. For example, in the named address space support that is the subject of another paper, the machine in question only needed one extended address keyword. It didn’t need the full generality of having multiple named addresses, but we added the more general support called out in the report that described named address because it might be needed for future ports.

• One ironclad requirement is in the machine independent section of the compiler, you cannot call into machine specific functions directly, but you must use the target hooks facility to do callbacks. If you only test on one platform, you would not notice that you are calling machine specific functions.

3.7 Fixing problems and not symptoms

GCC supports a wide number of architectures, ABIs, OSes, and languages. This improves code coverage and testing, but also makes testing and bug fixing more complicated. Where one could paper over a problem in a compiler with a more limited target, the design of GCC necessitates and the community enforces fixing problems, not symptoms. Sometimes the compiler has a segmentation violation or other internal error. Usually this is a problem caused earlier in the code, and may only happen for some machines. It can be tempting to just put an if statement to avoid the particular situation that is failing, but it is often times better to do research on why the thing was failing, and fix it upstream if possible.

3.8 What chipset to tune for

The maintainers for toolchain ports that span multiple implementations of the basic chipset, like the x86, power, mips, and arm platforms have to decide what are the characteristics of the tuning parameters for the default machine if the user did not specify to compile for a specific machine. If a chipset is made by several different companies, then there is a delicate balancing act as to which defaults to use. Even if a single company makes all of the chips, the maintainers have to decide whether to favor the machines that are currently out in the field, or the next generation of machines. In general at every major release, the maintainers should go back and look at the defaults, and in general favor the newer machines over the older ones.

Another area that comes in is maintaining tuned libraries for each chipset. In general, there is a tension between having libraries for every single option, and the whether this leads to exponential in the size of the release and lengthening the time it takes to qualify a product on each different variant of the hardware platforms. Hardware companies tend to be on the side of tuned libraries while distributions tend to be on the side of a single library.

3.9 Testing

It is good practice to do tests as much as possible so that the number of changes is smaller, and easy to isolate the bugs. As we mentioned earlier, you should merge to the mainline and test on a regular basis. However, merging up to the mainline has its costs, particularly when the compiler is in stage 1 development, as mainline can be broken for some periods of time.

One practice is to fire off a build/check of the mainline source that the development branch is merged up to, and compare it to the development branch to see whether there are regressions. If the mainline fails, then you need to investigate whether it is a general failure, or whether it is just your particular environment that fails. If the branch fails, then you need to figure out where you need to improve your code.

In addition to testing the code on your own platform, if you have touched machine independent files, you need to test at least one other platform to make sure that your changes do not break other compiler ports.

3.10 Simulators

• When working on a new chip variant or new machine, work is usually done on simulators before the real hardware is available. The problem with simulators is that they are slow. Ideally you will only run the binaries for controlled tests on the simulator. If you run the usual make check under a simulated machine and run the compiler in that simulator, it may take months to get one run done for the simulator to simulate the compiler compiling thousands of test cases. One solution is to make the simulator run an OS with a simulated network
device, and use the remote support that exists in the dejagnu test infrastructure to run the compiler locally, but send jobs to the machine. Another approach is to invoke the simulator each time to run each command.

- For a lot of work, a simple instruction set simulator will allow the compiler writer to test code generation much quicker than using a more complex simulator that models the whole system enough so you can boot an operating system and/or model pipelines and cache states. For tuning you might need to use the more complex simulator.

- One thing that is important is that the simulator and compiler both be validated against the actual hardware. For instance, one of us did a port to an embedded machine via a simulator years ago, and both the person writing the simulator and compiler mis-read the manual in the same way (whether 16x16->32 multiply sign extended or zero extended). The upshot was the compiler was compatible with the simulator, but it was not compatible with the hardware.

- If you are using Linux as a development system, you can register to use a simulator based on the magic numbers in the binary via /proc/sys/binfmt, so that for the purposes of running the testsuite, it looks like you are running a native executable.

- If you don’t have a good simulator available and the hardware is not yet available, consider writing a simulator based on the ones in the gdb package. The advantage of using those simulators is they provide a debugging environment in addition to pure instruction simulation.

- Another advantage of doing the gdb simulators is it lets other people run your code on their systems. This can be a helpful for instance if you have a question and need somebody to help you debug your compiler, or if you want to split doing development among different people or organizations.

3.11 Bringing up GCC on new hardware

- In terms of corporations and new hardware, one issue that comes up is how much access compiler developers have to test their code on the real platforms. Usually, these machines are fairly scarce and there are a number of people in the company trying to get access to the machine. If the development is being done by an outside group, it can be even more problematical to get enough access to the machine, particularly if the chip is late. This is a management issue, and is one of the things that corporations need to include in their plans is enough time for testing.

- Another thing that comes up is that alpha/beta hardware may undergo several cycles as the hardware is debugged. It is important that compiler developers be given access to the newer chip versions as they become available. Sometimes the changes are changes in what instructions are available (and the encoding), the instruction speed or cache policy between the alpha version of the chip and the chip that eventually reaches the customer. If a toolchain is optimized for the alpha chip, it may need to be re-optimized for the final chip.

- In addition to timing differences, alpha hardware often times has bugs in it, and a compiler team is called on to make the compiler not do some instruction sequences for alpha chips. One of GCC’s strengths is that it can easy to add such tweaks in a timely fashion and it would allow for other software to be tested on the machine until the next version of the chip comes out that fixes the bug.

3.12 Using GCC to perform what if type experiments

One area that hardware companies don’t use as much as they could is a collaboration between the compiler writer, machine architects, and simulator writers to explore what if type possibilities. For example, you might want to consider whether it is profitable to add a floating point multiply and add (FMA) instruction to the architecture. A private version of GCC would be made to add the FMA operation, the simulator would be modified to add FMA’s, and then all of the usual benchmarks would be run to see how frequently the FMA is used, compared to the cost of adding it to the next generation of machine. If it doesn’t pan out, you go on to the next test, and you haven’t spent much time in terms of finding this out compared to the cost of designing and fabbing a chip. Modifying GCC in this way is usually faster than a native compiler, since usually GCC already has such
support for other ports, and it is a matter of getting the right definitions in the machine description.

For example, the FMA case is pretty easy to do in GCC, and it took a very short time to do. During the investigation, the actual instruction changed several times, and it was a simple matter to change a private version of GCC to track these changes. GCC was used to experiment with the PowerPC architecture before the ISA was finalized and GAS was used to experiment with the AIX 64 bit XCOFF file format.

By having the compiler staff early enough in the design stage, it can avoid adding instructions that are hard to optimize for in the general case.

### 3.13 Embedded ports

If you are developing the compiler toolchain for an embedded environment, it is useful to have a complete setup that includes a stub library, so that you can run the test suites before you get a real chip made. Having a public simulator means that it is possible for other people to build and check for failures.

For embedded compilers, one development method that is done (sometimes with multiple people in parallel) is:

- Do the GNU binary utilities port for the assembler, linker, and disassembler.
- Do enough of the compiler port to start on the libraries.
- Add support for a simulator of the machine with system calls to do read, write, and exit.
- Port newlib or similar small library to work in the simulator to provide the full library that uses your stub system calls.
- Start work on the debugger.
- Finish up the compiler port.
- Contribute each of the changes back to their respective repositories.

### 3.14 Mailing lists, IRC, and firewalls

In the old days of GCC development, all work was done via the mailing lists (http://gcc.gnu.org/lists.html). Now in addition to the mailing lists, there is the #gcc channel on the internet relay chat irc.oftc.net/6667. Many of the developers hang out there, and it is useful to ask quick questions. Often times, questions are answered within minutes of being asked, that never go to email these days. If you are not able to use IRC, you are cutting yourself off from a valuable resource, and making your job harder.

If you use corporate mail systems like Notes or Outlook, it can mangle patches, and make it hard for people to communicate with you. Some people use mail systems like gmail for their mail, rather than their corporate mail. However, some companies prevent employees from using such services, and it can be a nightmare for people to do development with such mail systems. We have seen people have to leave the office so that they could get access to their mail or ftp from their home system to sites that might be blocked by the corporate firewall.

One thing to watch out for is whether your mail system is robust enough to receive the amount of mail in the mailing lists, and that you have enough space in your mail quota so that you don’t have an interruption in mail service. If the free software foundation mail servers get too many bounces, you will be removed from the lists. It can be really hard to track down why mail is disappearing when you work for a company with a large infrastructure. Another thing to check is whether mail from other developers is being considered spam and not being delivered to your inbox.

Developers need access to the machines that the sources are on to do checkouts and checkins of the source. This may involve setting up a hole in the firewall for accessing the gcc.gnu.org machine over port 22 (ssh) or it may involve having two computers, one on the internal network, and one on an external network that does not include anything that is confidential.

IRC and chat is another thing that is often times blocked by corporate security. At one company, IRC was blocked by company policy, and the GCC group had to get special access to get to the GCC IRC channel. This took a lot of justification to get this access. The point is the Gnu compiler is a joint effort among people
throughout the world, and if a company hampers its employees from participating in the development, they will be marginalized and have a harder time to get their code accepted.

3.15 Big bang vs. incremental changes

If possible, it is better to do changes incrementally rather than as a whole big change set. The reason is it is easier for the patch reviewers to grasp what you are doing as a whole, rather than trying to understand a couple of megabytes of changes. However, if the development tree is in stage 3 for a long time, these patches build up.

Now to be honest, a lot of work is done in big bang style, because of the patches building up. Also, the code can vary quite a bit as the developers are trying out different things. In fact as we write this paper, we have two major branches (named address spaces and power7) that need to be merged. Having a whole ton of patches to contribute all at once is going to make getting this code in harder. Usually, we try to make smaller patch sets to reduce the load on the reviewers, but sometimes when you make a lot of changes, it is hard to make small patch sets.

3.16 Getting help

It can be hard coming up to speed on working on the GCC compiler. There are various how-tos and documentation to read, but once you’ve read the documentation, it can be useful to ask people questions via the mailing lists or IRC channels. Generally, it is better to ask a specific question, rather than asking generalities. Remember, that most of the experts are very busy, and may not know all of the details of your particular implementation or port. Show that you’ve done some investigation of the problem rather than just saying something is broken, please help.

4 Getting your work committed

Once you’ve gotten through the legal hurdles and the challenges to actually add a new feature or fix a bug, the next step is to get the code committed to the tree. Sometimes this is a quick process, and sometimes this is an involved process. For example, it took 3-4 years to add support for having an option that the port could tweak to say that int bitfields are unsigned instead of signed.

4.1 Who needs to approve your code

Who needs to approve your changes depends on what areas of the compiler you have modified. Currently GCC has maintainers for various sub-parts of the compiler, and if you modify code in that section, you need the approval of the appropriate maintainer(s), or the global reviewers. Unless you are a maintainer for a section of code, you need approval for changes (which includes the global reviewers for their own changes). If you modify multiple areas of the compilers, you need the approval for each of the maintainers that your code modifies. As of this writing, there are maintainers for:

- CPU port maintainers (x86, arm, etc.)
- OS port maintainers (linux, hp/ux, rtms, etc.)
- Language front end maintainers
- Library maintainers
- Infrastructure and optimization maintainers (gimple, tree, rtl, non-algorithmic, etc.)
- Global reviewers approval are needed in a few cases that are global in scope, like adding a new port

If you are working on a GCC port for a new machine, you should go through the paperwork of being a maintainer for the machine.

4.2 Review cycle

Once you have finished the changes, you need to go through the steps of getting your code reviewed. The details of doing the review is covered in http://gcc.gnu.org/contribute.html. What we want to talk about is ways to make the process less painful.

- Make sure you have your paperwork up to date before submitting patches.
- Make sure the patch applies to the current tree before submitting the patch.
- Make sure you indicate what level of testing you have done, and if the change involves machine independent changes whether you have verified that the changes do not affect other ports.
• As we mentioned earlier try to submit patches in smaller self-contained chunks. All of the reviewers are busy people, and may pass on reviewing large patches, because they don’t have the time to look at a large patch in great detail. Note that there are posting limits for patches, and even mega-patches might have to be broken up so that you can submit them.

• Try to understand what the maintainer is saying and see where they are coming from. If you are unsure, ask the maintainers for suggestions, but don’t expect them to write or debug your code for you.

• Don’t take things personally. Even if you feel a reviewer is being unfair to you, keep things at a professional level when you submit patches. One of the things we have mentioned to people that we are mentoring is that it is helpful if you grow a thick skin and let the little things slide away. If you do feel that things are in a rut, consider trying to get somebody else with less history to work with the maintainer to find an acceptable solution.

• Ping the maintainers on the mailing list or IRC if the patches have sat around for awhile. It would be nice if all patches were acted on immediately, but with the great number of patches going out there, it is very easy for things to fall through the cracks. However, realize that the GCC development is done worldwide, and all of us have lives, so you need to wait a few days before sending out a ping message.

• Don’t use Notes or Outlook to submit patches. They tend to do weird things to the average patch, and make them impossible for a maintainer to try out the patch on their own system.

4.3 Coding style

One of the things that often times trip people up is not using the appropriate coding style. Just sit down, and learn the coding conventions (http://gcc.gnu.org/codingconventions.html) until it becomes second nature. This includes making sure all lines are less than 80 characters, no space in front of unary operators, and making macros uppercase.

Make sure you include a properly formatted ChangeLog file(s) for each subdirectory that has ChangeLogs. These ChangeLogs should describe all of the changes made.

If you are fixing a bug that is not in the test suite, add one or more tests to verify that the bug is fixed.

4.4 Who are you? Will we see you again?

• One of the common complaints about the current system is that people post patches and never get a review of the patch. Part of the problem is it can be hard to get noticed. In general, you increase the chances of people replying to you by being doing more work. However, if you are perceived as not knowing what you are talking about, people can sometimes ignore your posts.

• One suggestion is to spend some time fixing bugs is a good way to get your feet wet. Not only does it give you skills in working on the compiler, it makes you more visible, and maintainers often times remember frequent posters.

• One thing that people working for corporations don’t always understand is that fixing bugs other than just in your port, can raise your status higher, rather than fixing only bugs in your port. This is a feedback cycle, and as you gain more status, it may be easier to get people to review patches. Unfortunately, it is human nature to only concentrate on you little part of the compiler, and a lot of us don’t stick our necks out.

• If you are fixing a bug to get experience, and the bug becomes too complex, go to the next bug and look at that. We have seen people spend a lot of time on bugs like this, and it can be hard to justify spending that much time if it is too complex. One suggestion is to look at bugs that have been sitting around for awhile are likely not being worked on actively by somebody else, as it can be frustrating to work on a bug for a week, only to have somebody who knows that part of the compiler fix it in a small amount of time.

• Going to the GCC summit or other places to meet developers in person can often times help new developers get recognition, which in turn helps getting their patches noticed and reviewed. For senior developers, it can allow them to plan with other developers what to do in the future. In these days of
severely restricted travel budgets, it can be a hard sell to upper manager to balance the terms of the cost of the trip vs. the expected gains.

5 Working for a corporation

Most of the previous parts of this paper centered on the culture clash between free software developers and corporations from the point of view of the corporation and their employees. In this section, we will cover some of the issues that have come up over the years for free software developers when they work for the corporations. We should note that every company is different, and that these notes pertain to various companies over the past 20 years.

5.1 Getting a corporation to understand and support the GCC toolchain

If a company does not have a strong presence already in the free software world, it can be a really challenging task to get the company to look at GCC. The employees that are proposing to add GCC support really need to develop a strong business case to present to upper management which outlines risks, costs and return on investment. In other words, you need to think like a corporation, and submit these things in the language that the corporation understands. In most companies this involves getting a champion at the executive level who understands the issues and can write the business case. Often times in large companies you need to work up the layers, first convincing your local management and then middle management, and then executives.

Many years ago, a company was coming out with a new machine using a new chipset. At the time, the company had some serious issues with the compiler vendor they were dealing with, to the point that they were looking around for another compiler to use. A developer came across GCC at this point, and saw that somebody had started a port for the chipset they were using. The engineer mentioned this to his manager, and after some discussion about what was GCC and what really was covered by the GPL, the manager became a champion of using GCC, and was able to convince the senior vice presidents to go with providing support for GCC. Initially it was just thought of as the compiler to ship with their version of UNIX, and that customers would buy the normal compiler from the compiler vendor as an add-on cost.

When the port was finished, GCC became the standard compiler. At one particular point, the development team was told that GCC was much slower at a particular benchmark (dhrystone) than the other compiler, and that management was thinking of going back to that vendor. After some checking, it was discovered that the other compiler optimized the string copy (strcpy) of a constant string to be a fixed length memory copy (memcpy), but other than that, GCC was in general a better compiler. When the same optimization was added to GCC, it got much better numbers.

There are two take-aways from this story. The first is the importance of educating your management chain so that they can champion the support of GCC. If you can’t present a compelling case for supporting GCC, then the corporation might not support GCC. The second is not to rest on your past accomplishments, but you need to figure out what is needed (a new feature needed by the customer, faster benchmarks, etc.) and then implement it in a timely fashion. In a business context, this means scoping out the work, and then doing it successfully, and keeping to the schedule that you signed up for.

5.2 Full time employee, independent contractor, or non-work toolchain support

There are three main ways that a corporation might interact with a free software developer:

- If you are a full time employee who is tasked to do some work in the free software world. It might be a small part of your overall job, or it might be your full time job. Typically this is done when the company is committed to supporting GCC in some fashion or another.

- If you are an independent contractor or work for a independent company that does work for hire for the company. Often times this is done when a company wants a specific job done, but does not want to pay for people to interact with the free software community all of the time. The assumption is the contractor does all of the interaction about getting code committed, etc. but like in any business setup, there needs to be agreements about ownership of the code, who donates it, etc.
If you work for the company doing something not related to free software, and you work on free software on your own time. If you go this route, you probably need to get the appropriate paperwork that will allow you to contribute changes.

For the full time employees there tend to be a few different categories people fall into:

- **Lone wolf**: This is where there is a single person tagged with doing free software support. It can be important to make sure your company understands why they are doing GCC support and what benefits they expect to get.

- **Developer on a team**: For most developers, this is just like any other team job within the company, with the exception that you have to deal with some more issues about getting your code accepted.

- **Free software evangelist**: In this role, you might still be a developer, but your role is to help shape how the company deals with free software in general.

- **Manager or executive over free software developers**: In this role, you are likely to have to justify the budgets for the free software support, and make sure your employees can do the job you set out for them.

### 5.3 Other compilers

Most hardware companies support multiple compilers, where GCC is just one possible compiler. Sometimes it is a compiler created by the hardware company and sometimes it is a third party compiler where the hardware company works with the compiler vendors. Usually these compilers are in competition with each other for the hearts and minds of the corporation and the users. We take the view that competition is good, and that users benefit from having a choice. We might not be able to make GCC always the best choice, but we try to keep it in the ball game.

### 5.4 Secrets

If you have signed non-disclosure agreements with your company, you need to be careful about the leakage of information via source code in the repository, in mail messages, and on relay chat lines. As we mentioned above, you need to make sure that you need to schedule when new instruction set variants are made public before contributing the code. For example, scheduling the announcement of new instruction ISA before the closure of stage 1 of the compiler, so that the code could be contributed.

### 5.5 Learning to think like a corporation

- Just like the free software development process has its own ways of doing things, each corporation has its own ways of doing things. To be successful in a corporation, you need to learn how to communicate effectively in that setting, and to think in terms of risks, costs, and how what you are doing matters in the big scheme of things; or, in corporate jargon, profit-and-loss, risk-reward, and expense-to-revenue ratio. Think of a corporation as another puzzle, it is another complicated system to be reverse engineered or hacked.

- Do not approach upper level management with a general complaint. People who have been promoted to upper management are very operations-oriented. They want to know what action you want them to take, not presenting them with a general problem and hoping that they invent the solution. If a hacker does not know what the executive can or cannot do, that is okay. They would prefer a strawman proposal that they can accept, reject, or modify.

- Corporations operate on profit and loss. Find out what executive, probably a general manager, that is responsible for profit and loss in your line organization and his or her goals. Either align an existing project with those goals or convince the executive that a project will help with those goals. Regardless of departments and managers and task forces, projects that help the person responsible for profit and loss will not lack resources.

- Talk is cheap; follow the money. Executives consider the cost of a request or action. An executive stating that they like / support some project or some project is great / important incurs no cost. Strategies and reports and white papers and general platitudes carry no cost. The primary question of importance is a request for resources. To understand
what is important to an executive or organization or company, look at how and where resources are deployed.

- If an executive directs the hacker to the corporate lawyer or directs the hacker to assist the corporate lawyer with risk analysis, that is a brush off and busy work. There always will be legal risk in any action. Referring to the lawyer due to risk is a polite cop-out. The solution is not to try to resolve the legal concern, which is impossible. Instead, one must find the business justification (corporation will increase revenues X% or will lose an important customer bid). With the right business justification, the executive will decide the benefit outweighs the risk, accept the risk, and, if necessary, direct the corporate lawyer to figure out how to mitigate the risk. That is the preferred solution and the way to think like a corporation to get things done.

5.8 Educating your management

When our management changes, we often times find it takes some time to bring the manager up to speed of the ins and outs of contributing code to GNU. This paper is an attempt to record these thoughts for use in the future.

5.9 Out of sight, out of mind

In some companies, there is only one developer who is hired to do the GNU work, and the developer is off in their own little world, while the other developers are working on the main product. It can be lonely if there is nobody else doing similar work. How you deal with the issue is up to you. Strategies include, hanging out on the IRC channels, striking up work relationships with other developers even though you don’t work on the same things, or finding solace outside of your job.

Unfortunately, in these times if you are seen as disjoint to the rest of the company, it may make you the target of the next round of layoffs. Particularly if you are working at a remote location or from your home, you need to make sure that you are providing value to the company and that management knows what you are doing. You always need to keep in mind how what you are doing ultimately furthers a corporation’s goals. If layoff time comes and your boss doesn’t have an immediate answer for what have you done recently that will affect the company’s bottom line, then it may be time to update your resume.

Similarly if you are higher up in the management food chain or you are a single developer in a small company, you probably need to keep a current business case of why supporting GCC is important for your company, and why they should use you to do the job instead of expecting somebody else to do it for free, and update it regularly.

Nobody owes you a job, it is up to you to provide value to the company by the work that you do.

5.10 Monolith vs Fiefdoms

From the outside, corporations often times look like giant monoliths, speaking with one voice. However, from the inside, you often times find that you have many little fiefdoms that are competing against each other for resources and ultimately customers of the corporation.
5.11 Burnout

Doing too many all night sessions can lead to burnout. It is important to learn to deal with this before you do something that endangers your job. Part of dealing with burnout is maintaining the motivation for the job. Having outside interests can also help maintain your spirits at work. Finally, it is important in making schedules that you are realistic about the amount of time it is going to take you to do the job so that you do not have to do the all nighter because you underestimated the amount of work. If you do have long sessions where you are making a release, be sure to plan some downtime afterwards to allow you time to catch your breath.

5.12 Career Advancement

For many people, working on Gnu stuff can reduce the opportunities for career advancement, just because there are so few slots available strictly in terms of working on the toolchain. For a lot of developers that we know, this isn’t a problem, in that they are happy where they are, and don’t have loftier ambitions. If you do want to expand your role, often times you need to be creative and engineer the job you want at the next level.

In some companies, the usual career advancement is through the management ranks. People management is a different skill set than code development, and you need to judge whether you have that skillset or can grow into it. Not every developer wants to give up code development for writing presentations, working on spreadsheets, and managing people full time.

6 Summary

Some final thoughts:

- Make sure the legal hurdles are taken care of.
- Interact with other developers to form a personal relationship.
- Keep things at a professional level.
- Pay attention to the details and the correct procedures.
- Keep things up to date, and watch for new failures.
- Try to make smaller code submissions rather than giant all-in-one submissions.
Abstract

Multi-cores and multi-processors became ubiquitous during the last few years, and the trend is to increase the number of simple, power-efficient, and slower cores per chip. One of the results is that the performance of single-threaded applications did not significantly improve, or even declined, on new processors, which heightened the interest in compiler automatic parallelization techniques.

Our objective is to develop a framework in GCC to transform loops into pipelines of concurrent tasks using streams to communicate and synchronize. This transformation can either rely on user hints (pragmas) or on static analysis of control and data dependences. Our focus is to fully automate this transformation, which requires an integration with the Graphite polyhedral loop optimization framework and will lead to the development of runtime optimizations based on this polyhedral representation. This approach is complementary with the existing parallelization passes. This paper presents the steamization technique, the optimizations it enables, the interaction with other optimizations, and the necessary extensions to Graphite/PCP (polyhedral compilation package) for its integration.

1 Introduction

Increasing clock speed as well as micro-architectural and compiler advances have allowed steadily improving performance of single-threaded programs for many years. However, this does not seem to be the case any longer. The excessive design complexity and power constraints of large monolithic processors made this paradigm unsustainable and ultimately forced the industry to develop chip-multiprocessor (CMP) architectures, in which multiple processor cores are tiled on a single chip. As the number of transistors per chip continues to grow exponentially, the current trend is towards providing an increasing number of simpler, more power-efficient, and slower cores per chip. The performance of single-threaded applications is therefore expected to stagnate or even decline with new generations of processors. The applications need to exploit the multiple hardware execution threads available on these architectures to improve performance.

Besides the parallel applications, already covered by GCC’s automatic parallelization pass, parallelization often requires enabling transformations such as privatization, which allows removal of false dependences (write after read and write after write). However, this transformation often comes with a very high cost both in the amount of memory required and in execution time. The common privatization technique in which dynamic single assignment properties are achieved may even be impossible in some cases because the hardware resources are finite. Streamization allows reduction of the amount of memory necessary for privatization (compression after the expansion of memory). Though this comes with a reduction of the amount of parallelism exploitable, it also brings some benefits to the way communication through shared memory happens. The main objective of streamization is to optimize privatization.

Another objective of streamization is to achieve pipeline parallelism, which is complementary with data parallelism. It consists of generating pipelines of concurrent tasks communicating and synchronized through streams [12], which behave as blocking FIFO queues. We will elaborate on the specificities of our stream implementation in Section 2. One of the salient problems for parallelization on current architectures is to get data at the right place, at the right time. This problem
becomes increasingly acute with each new generation of processors as the memory wall builds up. Despite the fact that the clock speed of new processors is no longer increasing, more processing units become available, sharing the same limited resources. The number of CPU cycles available on CMPs per unit of time is increasing faster than the memory bandwidth because a single data bus has to feed more processing units. To fully exploit the available resources, applications need higher arithmetic intensity. Also, cache effects (in particular in the case of multiple hardware threads sharing cache lines) make scheduling decisions ever more complicated. We will try to show that pipeline parallelism can improve the behaviour of parallelized applications in that regard.

The paper starts by presenting the details of the stream communication library in Section 2. Section 3 presents the application of streamization to pipeline parallelism, while Section 4 shows how streamization can be used to optimize privatization. In Section 5, we describe the techniques we use to ensure that streamization does not inhibit other important optimizations such as vectorization or automatic parallelization. Section 6 presents the integration of streamization with Graphite. Finally, in Section 7 we evaluate the benefits of our approach with regards to previous contributions.

2 Privatization and Streams

Privatization is a technique used for eliminating false (or storage-related) data dependences (write after read and write after write dependences), in order to expose parallelism or enable program transformations. This technique consists in duplicating some memory area, with various levels of duplication. Depending on the context, it means making some memory area private (or local) to a thread, or to a point in an iteration domain, or even reaching a dynamic single assignment (DSA) property.

Let us consider the following example:

```c
int a;
for (i = 0; i < N; ++i) {
    a = ...;
    ... = ... a ...;
}
```

If we wish to parallelize this loop on P threads, it is sufficient to make a private to each thread. This means that P copies of a are necessary and we can distribute the iterations of the original loop among the threads. One possible solution, with the outermost loop fully parallel, is:

```c
int A[P];
for (k = 0; k < P; ++k)
    for (i = k; i < N; i += P) {
        A[k] = ...;
        ... = ... A[k] ...;
    }
```

Here the resulting code still has false dependences (not DSA), but they do not span across multiple outermost loop’s iterations. If our objective was to distribute the original loop, then we need to make more copies of a, one for each point of the iteration domain:

```c
int A[N];
for (i = 0; i < N; ++i)
    A[k] = ...;
for (i = 0; i < N; ++i)
    ... = ... A[k] ...;
```

There are no more false dependences (DSA code) and the iterations of the resulting loops are parallel. This type of privatization is also referred to as scalar expansion (or array expansion if a was of higher dimension).

In the rest of the paper, we will generally refer to scalar or array expansion as memory expansion, while privatization will be used as the general notion of making at least enough copies to enable a transformation. When more copies than necessary have been created (e.g., by memory expansion), reducing the number of copies will be referred to as memory compression.

2.1 Stream Communication Library

The stream communication library provides a simple interface for using streams, which are blocking FIFO queues, for communicating between two threads. The communication is unidirectional and ordered. The interface provides simple access operations to the elements in the stream, a push operation for inserting one element at the end of the stream, and a pop operation for removing the first element in the stream. Some more complex operations are also implemented for optimization purposes.

The notion of stream is not new and often has different accepted semantics. In some cases, streams behave as bags of elements in which no order is enforced on the access operations. This is notably the case in accelerator-oriented streaming languages (e.g., Brook or CUDA)
because the accelerator often does not provide for synchronization.

This library is not meant for programmers’ use, but as a target for code generation, in particular in the *Graphite codegen* pass\(^1\). In this case, streams are used to replace arrays and thus compress memory (counterpart of memory expansion or privatization), replacing the strong single assignment property by a relaxed single assignment in which memory is reused after a synchronization point that ensures the overwritten data is no longer useful. We will elaborate on this use of stream communication in Section 4.

### 2.2 Simplified Stream Semantic

To make the use of streams straightforward in optimization frameworks such as the polyhedral compilation package (PCP), we need to provide a simple approximation of the stream semantic. Though this approximation does not correspond to the underlying implementation, it should be restrictive enough to ensure the correctness of the generated code.

In most cases, streams will simply be considered as infinite arrays in which elements are only written and read once, in sequential order, like a FIFO queue.

As data is duplicated and, in this simplified semantic, each element is assigned only once, we will consider privatization through streams to provide DSA.

Such a semantic clearly goes against the objective of reducing the memory used for privatization and is not implementable with finite hardware resources. We need to impose some restrictions on the communication patterns to allow reusing memory.

### 2.3 Implementation and Interface

The implementation of streams is in the form of directional channels of communication that behave as blocking FIFO queues. The producer enqueues elements into the stream and the consumer dequeues them. Streams are implemented as circular buffers to avoid excessive memory usage, but the buffer can be dynamically resized if this appears necessary. The blocking behavior means that the queuing operation blocks when the buffer is full and the dequeuing operation blocks when it is empty.

Stream communication serves two purposes: first, it privatizes the data, thus removing any output or anti-dependences; second, it relaxes the flow dependences as it decouples the producer from the consumer. Because the stream operations have blocking semantics (*i.e.*, the producer waits until there is free space in the stream and the consumer waits for elements in the stream), the streams also provide synchronization between the producer and consumer tasks.

The implementation of streams without *ad hoc* hardware support presents two principal sources of overhead. The first is the need to synchronize read and write operations, which can be very expensive. The second problem comes from bad cache behaviour due to false sharing when the producer and consumer access elements that are close together. Because the stream elements are stored in a circular buffer, this happens when the buffer is almost full or almost empty.

Both of the sources of overhead can be almost completely mitigated by adjusting the granularity of communication. The synchronization overhead can be reduced by aggregating multiple elements in blocks and only synchronizing the accesses to blocks of elements. The cache degradation can be avoided by preventing the producer and the consumer from accessing elements in the same cache line.

To increase the granularity of the communication, we introduce sliding windows (see Figure 1), in which the reads and writes to the buffer occur. These sliding windows are used for reducing the amount of synchronization, which is required only when the windows are sliding. The windows can also be aligned on cache boundaries to avoid false sharing. One or more cache lines are reserved for writing/reading to/from the stream.

The interface of the streaming library provides both basic access functions, like *push* and *pop*, and more efficiency-oriented functions that avoid unnecessary

---

\(^1\)For details, see the “Design of Graphite and the Polyhedral Compilation Package” paper in the GCC Summit 2009 proceedings.
Automatic Streamization in GCC

The basic interface provides the following simple access functions:

```c
void push (stream, element);
element pop (stream);
bool end_of_stream (stream);
```

The `push` and `pop` functions are the usual access functions to FIFOs, but it should be noted that both represent copies. The `end_of_stream` function checks whether the producer has finished working and the stream is empty. It should be called whenever it is not possible to test for termination in other ways. This is necessary, for example, when the producer loop is under dynamic control, so that – even at runtime – it is impossible to know the number of iterations until the loop finishes. In most examples in this paper, the producer and consumer iterate on the same domain, so there is no need for this check, but examples can be found in Sections 3.1 and 4.3.

The remaining access functions allow direct access in the stream buffer, one window at a time. To reduce overhead, a task can request an empty window from the stream and store the elements directly, using `get_tail_window`. Once the window has been filled, `commit_window` makes it available for reading by the consumer task. The same mechanism is available for the consumer. An illustration of the application of this technique can be found in Section 5.

```c
element *get_tail_window (stream);
void commit_window (stream);
element *get_head_window (stream);
void pop_window (stream);
```

These operations not only reduce the runtime overhead, they also avoid useless copies. They would also allow a seamless integration with software transactional memory [9] to enable speculative execution of consumer threads.

### 3 Exploiting Pipeline Parallelism with Loop Streamization

Loop streamization is a program transformation that enables pipeline parallelism in sequential programs. As with other similar techniques that we further discuss in Section 7, it relies on memory expansion (privatization) and synchronization. This technique is primarily based on loop distribution and software pipelining.

Our objective is to automatically enable and exploit parallelism in sequential programs while avoiding non-scalable expansion schemes. As we will see, streamization will allow us to explore the entire space of memory expansion, ranging from the original sequential code to the highest level of parallelism with full privatization (which is often non-realistic). The choice of the amount of memory duplication, and therefore of the amount of parallelism enabled, can be both static and/or dynamic.

#### 3.1 Streamizing while Loops

The first step in the streamization process is to partition the computation into tasks that present a producer-consumer relationship. In the general case, tasks will have flow dependences in between each other; otherwise, they are only bound by control dependences. The producer and consumer originally communicate through a shared data structure, in which the producer writes and the consumer reads. We replace this shared memory communication by stream communication. The blocking nature of our stream implementation implicitly synchronizes the execution of the producer and consumer tasks.

The static analysis involved in partitioning a loop into tasks is identical to that of loop distribution [11, 6]. After building the program dependence graph [10], the strongly connected components are coalesced. The nodes of the resulting directed acyclic graph can be partitioned with, for example, an iterated minimum cut algorithm because each cut edge will represent inter-thread communication. Another option is to try to statically balance the load of each partition using a sparsest cut algorithm with weighted vertices.

As an illustration of the task partitioning, consider the following `while` loop:

```c
S1:     while (data = read (input))
     {
     S2:            tmp = process (data);
     S3:         write (tmp, output);
     }
```

This type of loop actually represents a very common case. The `read` operation can be thought of as reading from a file in applications like video decoding or getting the next element in a linked data structure (list, tree ...). In most cases, this operation cannot be executed concurrently because it updates the state of the `input`
After coalescing strongly connected components, the dependence graph is reduced to the one shown in Figure 2. The same remarks hold for the write operation and its output parameter.

For this code, the dependence graph is presented in Figure 2. In this simple case, the coalescing of the strongly connected components only removes the self-cycles on S1 and S3, exposing the potential cuts.

The parallelization of this loop can be achieved either as a doacross schedule or by pipelining. The doacross parallelization schedules iterations of the loop on different threads and introduces synchronization for each cross-iteration dependence (we do not take into account output and anti-dependences because they can be eliminated by privatization [8]). The pipelining approach will schedule each strongly connected component in the dependence graph on a different thread and synchronize each inter-thread flow dependence [14]. This synchronization being implicit in streams, we would get the following streamlined pipeline:

```c
while (data = read (input)) {  
  push (S_data, data);
}
while (! end_of_stream (S_data)) {  
  tmp = process (pop (S_data)); 
  push (S_tmp, tmp);
}
while (! end_of_stream (S_tmp)) {  
  write (pop (S_tmp), output);
}
```

To understand the reason why pipelining is more efficient than other approaches, we show in Figure 3 the doacross and pipelined schedules.

If we compare the execution traces obtained by the two techniques, it is easy to see that pipelining will be more efficient because it shortens the critical path. The insight here is that the inter-core synchronization plus the communication of the data will introduce a high latency. This will be problematic if such latency is allowed on the critical path. In the doacross schedule, the dependence that is enforced across threads is a cross-iteration dependence, which means that the underlying memory location cannot be privatized. The synchronization and communication overhead must be paid at least once for each iteration of the loop. However, in the pipeline schedule, we keep such loop-carried dependences on the same thread (this comes from the fact that each strongly connected component of the dependence graph is scheduled on a single thread). The remaining dependences will still have the same overhead, but privatization will better tolerate (or hide) the latency.

### 3.2 Dynamic Loop Fusion

Another way to look at loop streamization is to consider that it is like a halfway position between distributed and fused loops. Depending on the amount of memory used in the stream buffer for duplication, the producer and consumer loops can be more or less coupled. Consider the following streamized loop:

```c
stream S;
for (i = 0; i < N; ++i) { 
  push (S, ...);
} 
for (i = 0; i < N; ++i) { 
  ... = ... pop (S) ...;
}
```

If the stream S only allows storage of a single element at
a time, the possible execution schedule of the two loops will be identical to that of the fused loops:

\[
\text{stream } S; \\
\text{for } (i = 0; i < N; ++i) \{ \\
\quad \text{push } (S, \ldots); \\
\quad \ldots = \ldots \text{ pop } (S) \ldots; \\
\}
\]

Such a fusion (as well as the replacement of the stream by a scalar) is always possible because the producer and consumer necessarily traverse the same iteration domain. On the other hand, if the stream allows an infinite number of elements to be stored at a time (or, in this case, at least \( N \) elements), then the possible schedules can be as decoupled as distributed loops.

3.3 Experimental Evaluation

We evaluate the potential of the streamization technique on a kernel extracted from the GNU Radio project [5]. This kernel was originally extracted by Marco Cor- nero, from STMicroelectronics, and further adapted for streaming by David Rodenas-Pico, from the Barcelona Supercomputing Center, for the needs of the ACOTES project [1]. We also slightly modified the kernel for our experiments, by annotating it with OpenMP task pragmas with firstprivate and lastprivate clauses. The main loop of the annotated kernel is presented on Figure 4. We will show, as a motivation for the optimizations under development, what can be gained from streamizing the code. We write the code as it would be generated by an optimizing compiler, with no additional manual optimizations. The implementation of the streaming library takes advantage of the memory hierarchy by aggregating communication in reading/writing windows. These windows should at least be of the size of an L1 cache line, which reduces false sharing and improves performance [12].

The OpenMP annotations we use constitute a minor extension to the OpenMP3.0 standard. We only introduce the association of the lastprivate clause on task constructs. The presence of this clause means that the corresponding task produces a value that needs to be propagated to the enclosing context, so any subsequent task will see this value. In other words, variables marked with lastprivate are produced by the task and variables marked with firstprivate are consumed. This knowledge allows to build pipelines of producer/consumer tasks.

The evaluation of this benchmark is performed using a modified version of GCC4.4 available in the streamOMP branch. The experimental results are presented on Figure 5.

The streamized code shows reasonably high speedups. On average, the execution of the hand-streamized code is more than three times faster than the sequential version on all platforms. Such results are a strong incentive to continue the development of the streamization framework in GCC. We note that the load balance is not perfect yet as only two of the thirteen filters present in the application have a high arithmetic intensity. This results in equivalent speedups on all platforms despite the fact that platforms 1 and 2 have 8 hardware threads whereas platform 3 only has 4 hardware threads.

4 Optimizing Privatization with Streams

To expose parallelism in a sequential program or to enable loop transformations, it is often necessary to remove false dependences by privatizing the memory locations involved in these dependences. Though privatization enables some optimizations, it can be excessively expensive, both in terms of memory requirements and execution time. We will first show how streamization can improve the memory requirements of privatization for the purpose of enabling an optimization.

4.1 Reducing the Memory Footprint

Privatization through memory duplication has the benefit of exposing the maximum amount of parallelism, but it also is the most expensive technique in terms of memory usage. To avoid an excessive increase in the memory footprint when there are no loop-carried dependences, a common technique consists of only making one copy per concurrent thread. However, this is not always possible.

Consider the following example:

```c
int a;
for (i = 0; i < N; ++i) {
    a = \ldots;
    for (j = 0; j < M; ++j) {
        a += B[j][i];
    }
    \ldots = \ldots a \ldots;
}
```
#pragma omp parallel
{
 #pragma omp single
 |
 #pragma omp task firstprivate (pair, fm_qd_conf) lastprivate (fm_qd_value)
 fm_quad_demod (&fm_qd_conf, pair.first, pair.second, &fm_qd_value);

 #pragma omp task firstprivate (fm_qd_value, lp_11_conf) lastprivate (band_11)
 nctaps_filter_ffd (&lp_11_conf, 1, &fm_qd_value, &band_11);

 #pragma omp task firstprivate (fm_qd_value, lp_12_conf) lastprivate (band_12)
 nctaps_filter_ffd (&lp_12_conf, 1, &fm_qd_value, &band_12);

 #pragma omp task firstprivate (band_11, band_12) lastprivate (resume_1)
 subtract (band_11, band_12, &resume_1);

 #pragma omp task firstprivate (fm_qd_value, lp_21_conf) lastprivate (band_21)
 nctaps_filter_ffd (&lp_21_conf, 1, &fm_qd_value, &band_21);

 #pragma omp task firstprivate (fm_qd_value, lp_22_conf) lastprivate (band_22)
 nctaps_filter_ffd (&lp_22_conf, 1, &fm_qd_value, &band_22);

 #pragma omp task firstprivate (band_21, band_22) lastprivate (resume_2)
 subtract (band_21, band_22, &resume_2);

 #pragma omp task firstprivate (resume_1, resume_2) lastprivate (ffd_value)
 multiply_square (resume_1, resume_2, &ffd_value);

 
 #pragma omp task firstprivate (fm_qd_buffer, lp_2_conf) lastprivate (band_2)
 nctaps_filter_ffd (&lp_2_conf, 8, fm_qd_buffer, &band_2);

 #pragma omp task firstprivate (ffd_buffer, lp_3_conf) lastprivate (band_3)
 nctaps_filter_ffd (&lp_3_conf, 8, ffd_buffer, &band_3);

 #pragma omp task firstprivate (band_2, band_3) lastprivate (output1, output2)
 stereo_sum (band_2, band_3, &output1, &output2);

 #pragma omp task firstprivate (output1, output2, output_file, text_file)
 |
 output_short[0] = dac_cast_trunc_and_normalize_to_short (output1);
 output_short[1] = dac_cast_trunc_and_normalize_to_short (output2);
 fwrite (output_short, sizeof (short), 2, output_file);
 fprintf (text_file, "%-10.5f%-10.5f\n", output1, output2);
 |
}
}
}

Figure 4: Kernel extracted and adapted from the GNU Radio project [5] with OpenMP annotations. The lastprivate clauses on tasks enable streamization.

In this case, the loops are sequential and the array B is not traversed in the right order. Loop interchange is necessary to improve performance, but it is not possible in this imperfect loop nest. The first step is to expand the scalar a and distribute the outermost loop, then we can interchange the loops to get the following code:

```c
int A[N];

for (i = 0; i < N; ++i)
```
Platform 1: Dual AMD Opteron™ Barcelona B3 CPU 8354 with 4 cores at 2.2GHz, running under Linux kernel 2.6.18, and the following characteristics of the memory hierarchy:

- L1 cache line size: 64 B
- 64 KB per core L1 cache
- 512 KB per core L2 cache
- 2 MB per chip shared L3 cache
- 16 GB RAM

Platform 2: IBM JS22 Power6 with 4 cores, each two-way SMT, at 4GHz, running under Linux kernel 2.6.16. Memory characteristics:

- L1 cache line size: 128 B
- 64 KB L1 cache
- 2 MB per core L2 cache
- 8 GB RAM

Platform 3: Intel® Core™2 Quad CPU Q9550 with 4 cores at 2.83GHz, running under Linux kernel 2.6.27, and the following characteristics of the memory hierarchy:

- L1 cache line size: 64 B
- 32 KB per core L1 cache
- 2 independent 6 MB shared L2 caches
- 4 GB RAM

Figure 5: Speedups to sequential execution obtained on the GNU Radio kernel presented on Figure 4.

```plaintext
A[i] = ...;

for (j = 0; j < M; ++j)
for (i = 0; i < N; ++i)
A[i] += B[j][i];

for (i = 0; i < N; ++i)
... = ... A[i] ...;
```

Now the array traversal is correct and the innermost loop is parallel. If we want to parallelize the outermost loop, we cannot interchange (or we would just go back to an inefficient traversal of B), so we need to further privatize the array A. However, the expansion of array A does not parallelize the outermost loop (see below), and loop skewing will not allow for a higher granularity of parallelism than the innermost loop parallelization.

```plaintext
int A[M][N];

for (i = 0; i < N; ++i)
A[0][i] = ...;

for (j = 0; j < M; ++j)
for (i = 0; i < N; ++i)
```

```plaintext
for (i = 0; i < N; ++i)
... = ... A[M][i] ...;
```

If, instead of a scalar expansion followed by an array expansion, we use stream privatization, we can both reduce the amount of memory used and relax the synchronization. We obtain the following code:

```plaintext
stream S[M];

for (i = 0; i < N; ++i)
push (S[0], ...);

for (j = 0; j < M; ++j)
for (i = 0; i < N; ++i)
{ tmp = pop (S[j]) + B[j][i];
  push (S[j+1], tmp);
}

for (i = 0; i < N; ++i)
... = ... pop (S[M]) ...;
```

One interesting thing to note is that, while skewing allows to parallelize along Lamport hyperplans, streamization allows a much more relaxed wavefront parallelization schedule, as shown on Figure 6.
The amount of memory used to privatize the scalar $a$ is also reduced by streamization. As we noted in Section 2.3, a stream can be made to use as little as a single copy of the privatized memory area (the scalar $a$ here) or any arbitrary integer $\sigma$ (the size of the stream) multiple of this amount. In the original code, $a$ occupies $O(1)$ memory space. After the scalar expansion, the array $A$ used for privatization uses $O(N)$ and then $O(M \times N)$ after array expansion. The streamed version of the code uses $O(\sigma \times M)$ space, which can be significantly lower than its non-streamized counterpart.

4.2 Memory Duplication vs. Parallelism

A further optimization is possible on the previous example if we only make one stream copy per thread. Consider that the outermost loop will be executed in parallel over $P$ threads. Then it is not necessary to use memory expansion to privatize the streams used to communicate between the different iterations of the outermost loop. The following code would result from this memory compression:

```c
stream S[P];
for (i = 0; i < N; ++i)
    push (S[0], ...);

for (k = 0; k < P; ++k)
    for (j = k; j < M; j += P)
        for (i = 0; i < N; ++i) {
            tmp = pop (S[k]) + B[j][i];
            push (S[(k+1)%P], tmp);
        }
for (i = 0; i < N; ++i)
    pop (S[0]) ...;
```

In this final form, the privatization of $a$ only requires $O(\sigma \times P)$ copies. This is especially interesting because we can control the amount of parallelism available by increasing the amount of memory duplication at the finest granularity. If we increase the size of the streams $\sigma$, we can reduce the coupling between the concurrent threads, while $P$ directly controls the amount of parallelism available. This allows exploration of the full space of memory expansion, from scalar to multi-dimensional array expansion (if we make streams of size $N$ and $P = M$).

One interesting benefit of our approach is to allow us to very precisely choose the tradeoff between the amount of parallelism made available and the amount of memory used in the privatization process.

4.3 Postamble to Privatization

In most privatization techniques, there is a preamble (duplicating the memory) and a postamble (storing in the original memory the values that would have been generated in the sequential computation). This postamble is necessary if the memory is read afterwards or if it is not possible to determine whether it is still needed. This last step can be fairly expensive and complicated in some cases. However, the use of streamization based privatization makes this trivial. As streamization requires the sequentialization of the computation in the
pipeline, the last computed value for each memory location corresponds to the last value stored in the stream used for privatizing that location. It is therefore enough just to store back to the original memory location the last value of the stream, which is an $O(1)$ operation.

For example, memory expansion in the following code requires the last computed value of $a$ to be kept for the use statement after the loop:

```c
int a;
for (i = 0; i < N; ++i) {
    if (condition (i))
        a = ...;
        ... = ... a ...;
}
use (a);
```

If we privatize, we get the following:

```c
int A[N];
int a;
for (i = 0; i < N; ++i) {
    if (condition (i))
        A[i] = ...;
        ... = ... A[i] ...;
}

a = postamble (A);
use (a);
```

where `postamble` returns the element in array $A$ that was last assigned. If the loop was executed in parallel, then it is necessary to keep track of all stores to the array and find the maximum on the indices in the array where a store occurred. Though this operation is parallelizable, the operation requires $O(N)$ steps.

If the loop was streamized instead, it is sufficient to store in $a$ the last element in the stream:

```c
int a;
stream S;
for (i = 0; i < N; ++i) {
    if (condition (i))
        push (S, ...);
}
while (! end_of_stream (S)) { ...
    ... = ... pop (S) ...;
}
a = last_element (S);
use (a);
```

## 5 Interaction with GCC Optimization Passes

It is important to ensure that the streamization pass does not inhibit, or hinder the applicability of, other optimization passes in GCC.

Our objective is to ensure that if, for example, a loop is vectorizable prior to streamization, then the streamized loop also benefits from vectorization.

To achieve this, we introduced the window operations described in Section 2.3. Using these operations, we achieve some form of loop blocking in which the inner loop will present the same structure as the original loop, and which is therefore equally vectorizable.

Without entering into the implementation details, the stream implementation relies heavily on aggregation of multiple elements in windows (see [12] for details). This has multiple advantages, in particular for cache behavior and synchronization overhead reduction. In the following code, the streamization process does not access the stream element-wise, as we used in the other examples, but by blocks of `window_size` elements at a time. This allows the innermost loop, which iterates over each one of these blocks, to have a regular behavior conducive to vectorization.

We used this example in previous sections:

```c
int a;
for (i = 0; i < N; ++i) {
    a = ...;
    ... = ... a ...;
}
```

instead of the following streamized loop in which no further optimization is possible due to the access function calls:

```c
int a;
stream S;
for (i = 0; i < N; ++i) {
    push (S, ...);
}
for (i = 0; i < N; ++i) {
    ... = ... pop (S) ...;
}
```

We will have:

```c
stream S;
int *wl, *sw2;
for (i = 0; i < N; i += window_size) {
    wl = get_tail_window (S);
    for (j = 0; j < window_size; ++j) {
        wl[j] = ...;
    }
    ...
    ... = ... pop (S) ...;
}
a = last_element (S);
use (a);
```
commit_window (S);
}
for (i = 0; i < N; i += window_size) {
    w2 = get_head_window (S);
    for (j = 0; j < window_size; ++j) {
        ... = ... w2[j] ...;
    }
    pop_window (S);
}

The outer loop handles synchronization and communication, while the nested loop is very similar to what we could have obtained by expanding the scalar a and distributing the loop, as we did in Section 2. The important parameter will be the window size, which will determine profitability: a small sliding window would create too much synchronization, while a too-large window would make the processed data not fit in the caches.

In the fully dynamic case, the runtime may determine the size of windows, but at the expense of having to make the vectorization decisions at runtime and using a vector and scalar version of the computation task.

In the static transform case, a part of the synchronization is transformed into static control using loop blocking: this enables vectorization at compile time and eliminates some runtime checks. The window size chosen at compile time may not be the best because the memory communication and synchronization costs are less precise at compile time.

The stream runtime dynamically performs loop fusion and loop blocking, operations that may be performed at a lesser cost at static time by the Graphite framework, but with greater uncertainty on the dynamic costs.

6 Integration with Graphite/PCP

The data flow analysis of memory accesses is available in the polyhedral representation GPOLY of the PCP infrastructure. Some of the transformations performed by GPOLY involve data privatization that can be optimized using streams. GPOLY tags some of the arrays that have been used for privatization as streams when the data flows through the array as through a FIFO. PCP then annotates the dimensions of the arrays that can be compressed into streams, and the code generation produces the streamized code without further analyses.

6.1 Data Flow Analysis for Streamization

A stream can be used when the memory communication between a consumer and a producer has the following properties:

- Source and target iteration domains are equal: the number of points and the iteration order over these points are identical; and,

- Data dependences between producer and consumer are regular: the consumer must read the data in the same order it was produced.

Under these conditions the dimensions that can be contracted into a stream are marked with an annotation by PCP.

6.2 A Stream Extension to PCP

In PCP, the streams are represented as arrays with one of their dimensions annotated with the stream flag:

streamType <- array(10, 100 | stream(1))

This example defines the type of an array of 10 by 100 elements, in which the second dimension containing 100 elements could be compressed using a stream: the stream annotation can be ignored, in which case a full size array has to be generated.

6.3 Stream code generation from PCP

Streams are created in the Graphite code generation. The stream annotation allows the code generation to transform an array into a stream without further data flow analysis. There is no need to communicate the end of generated data between the producer and the consumer because the iteration domains in which they occur are identical. This, and the fact that a stream array is defined and used only once, allows the code generator to always insert the initialization and the finalization of the streams before the loop nest of the producer and after the loop nest of the consumer.
Stream programming has recently attracted a lot of attention as an alternative to other forms of parallel programming that offers improved programmability and may, to a certain extent, reduce the severity of the memory wall. Many languages and libraries are available for programming stream applications. Some are general-purpose programming languages that hide the underlying architecture’s specificities, while others are primarily graphics processing languages, or shading languages. Some hardware vendors also propose low-level interfaces for their GPUs.

The StreamIt language [2] is an explicitly parallel programming language that implements the Synchronous Data Flow (SDF) programming model. It contains syntactic constructs for defining programs structured as task graphs. Tasks contain Java-like code that is executed in a sequential mode. StreamIt provides three interconnection modes: the Pipeline allows the connection of several tasks in a straight line; the SplitJoin allows for nesting data parallelism by dividing the output of a task in multiple streams, then merging the results in a single output stream; and, the FeedbackLoop allows the creation of streams from consumers back to producers. The channels connecting tasks are implemented either as circular buffers, or as message passing for small amounts of control information.

The Brook language [3] provides language extensions to C with single-program multiple-data (SPMD) operations that work on streams (i.e., control flow is synchronized at communication/synchronization operations). Streams are defined as collections of data that can be processed in parallel. For example: “float s<100>;” is a stream of 100 independent floats. User-defined functions that operate on streams are called kernels and use the “kernel” keyword in the function definition. The user defines input and output streams for the kernels that can execute in parallel by reading and writing to separate locations in the stream. Brook kernels are blocking: the execution of a kernel must complete before the next kernel can execute. This is the same execution model that is available on graphics processing units (GPUs): a task queue contains the sequence of shader programs to be applied on the texture buffers. The CUDA infrastructure from NVIDIA [4] is similar to Brook, but also invites the programmer to manage local scratchpad memory explicitly: in CUDA, a block of threads, assigned to run in parallel on the same core, share access to a common scratchpad memory. CUDA is lower level than Brook from a memory control point of view. The key difference is that CUDA has explicit management of the per-core shared memory. Brook was designed for shaders: it produces one output element per thread, any element grouping is done using input blocks reading from main memory repeatedly.

The ACOTES project [1] proposes extensions to the OpenMP3.0 standard that can be used for manually defining complete task graphs, including asynchronous communication channels: it adds new constructs and clauses such as a new task pragma with clauses for defining inputs and outputs [7]. The implementation of the ACOTES extensions to OpenMP3.0 includes two parts: the compiler part translates the pragma clauses to calls to a runtime library extending the OpenMP library. The ACOTES extensions are an attempt to make communication between tasks explicit. Channels can be implemented on top of shared memory as well as on top of message passing. ACOTES extensions can be classified MIMD because several tasks can execute in parallel on different data streams. This aims to shift the memory model of OpenMP from shared memory to distributed memory for the task pragmas.

The resulting ACOTES programming model can be compared to the Brook language: these languages both provide the notion of streams of data flowing through processing tasks that can potentially contain control flow operations. The main difference between these two programming languages is in their semantics. In the execution model of a Brook task, the task is supposed to process all the data contained in the stream before executing another task. The tasks in the ACOTES semantics are non-blocking: the execution of a task can proceed as soon as some data is available in its input streams. The main limitation of the Brook language is the intentionally blocking semantics that follows the constraints of the target hardware (i.e., GPUs, where the executing tasks have to be loaded on the GPU, an operation that has a non-negligible cost). The design of the Brook language and of CUDA follow these constraints, restricting the expressiveness of the language, intentionally. The ACOTES programming model does not contain these limitations and the runtime library support of the ACOTES streams can dynamically select the blocking semantics of streams to fit the cost constraints of the
target hardware.

Another interesting approach to generate the data transmission towards the accelerator boards is that of the CAPS enterprise: codelets are functions [13] whose parameters can be marked with input, output, or inout. The codelets are intended to be executed remotely after the input data has been transmitted.

The technique that is closest to our approach is the decoupled software pipelining (DSWP) [14] proposed by Rangan et al. The authors extract parallelism by building the program dependence graph, then isolating in separate threads the strongly connected components of the graph. They rely on hardware support in the form of synchronization arrays and evaluate their code on a simulator. They recognize that, without hardware support, their technique only results in slowdowns. The static analysis used in this framework is unable to handle cases other than loop distribution.

8 Conclusion

We presented some motivating factors for the extension of the Graphite and PCP optimization infrastructures with streamization. Streamization exploits pipeline parallelism in otherwise sequential loops to reduce the memory used for privatization and to finely explore the tradeoff between parallelism and memory expansion.

The paper details the interactions of streamization with other GCC optimizations and suggests an extension to PCP for integration with Graphite.

References


Abstract

Tracepoints in the GNU Debugger (GDB) are an advanced feature that collects data from the program without stopping it, thus enabling GDB to help debug time-critical code. CodeSourcery has been developing GDB support for a high-performance telecom application. The application runs under Linux, but the usage model includes diagnosis of systems in the field, so we have built a debugging stub that links into the app and controls it from a dedicated thread. In addition to regular debugging (both all-stop and non-stop), this stub supports tracing.

On the GDB side, tracepoints are being made into a type of breakpoint, and the remote protocol dependency is being eliminated by moving tracing operations into GDB’s target vector. In addition to reducing old idiosyncrasies, these changes enable new features that are also being added, such as conditional tracepoints, and reading of trace data from a file. In addition, we are adding additional tracepoint features, such as trace state variables, disconnected tracing, and Machine Interface (MI) support.

2 Background

GDB[1] is the debugger of choice for the GNU project[2]. Recently we implemented support for non-stop debugging[4], which reduces the debugger’s impact when working with multithreaded programs. In non-stop debugging, the user can choose to stop only threads of interest, while other threads continue running.

However, for some cases even this behaviour is too intrusive:

- Threads often communicate with each other, so stopping a thread to examine its state may perturb other threads such that a defect no longer appears.

- A bug may manifest only rarely and it is necessary to observe the system for an extended period of time.

The traditional solution is to insert print statements into the program at suitable locations, rebuild the application and then rerun it. Many programs have expanded this approach into elaborate logging facilities, but it has limitations: the program needs a place to print to (an especial difficulty for embedded systems), and the printing itself can affect program timing—the I/O subsystem effectively serializes prints coming from multiple threads, since they all go into a single output stream.

A more sophisticated strategy extends the debugger’s breakpoint capability. Instead of simply stopping a thread when a breakpoint is hit, the debugger gathers and records data, then resumes thread execution. Later the user can go back and examine the recorded data. In GDB this feature is called tracing, and the recording points are called tracepoints. They were added in 1999[5], although they have not seen much use since then, as tracepoints require considerable target-side support (collectively referred to as the remote agent or simply agent), on the order of several thousand lines of code, and including a small bytecode engine.

Nevertheless, tracepoints seem so valuable to high-performance computing that we have taken them up again, modernizing the code, adding additional tracepoint features, and making some long-overdue changes in the architecture.

2 Target System

The target system for this work is a high-performance phone switch built around multi-core x86-64 processors. While the operating system is Linux, and the actual telecommunications processing is done by an application program running under Linux, it is not plausible
to run GDB on the target, nor even gdbserver; one of
the planned usage models is to connect to systems al-
ready out in the field, and examine their operation with-
out stopping the application, even for a fraction of a sec-
ond. So a ptrace-based strategy is not acceptable.

Instead, we implemented a debugging stub that is linked
into the application. Unlike the traditional GDB stub
that takes control from the application code and then
hands it back, in a sort of coroutine arrangement, this
stub runs in a dedicated thread, stopping and starting
the other threads using signals delivered via
`pthread_kill()`. Breakpoints are still implemented with traps; `SIGTRAP`
is caught by the thread’s signal handler, which not-
tifies the debug thread and then suspends itself with
`sigsuspend()`. Multiple threads may raise signals
simultaneously, so there is a bit of complexity in pre-
venting deadlocks and races as the debug thread simul-
taneously installs breakpoints here, single-steps a thread
there, and responds to packets arriving from GDB in the
meantime.

For library and system services, the debug thread relies
on a combination of normal Unix calls, and API pro-
vided by the application. For instance, reads and writes
in the application’s part of the address space must al-
ways go through API. The application gives the stub
only a single block memory to work in; we then use
Doug Lea’s dlmalloc to manage that space efficiently.

The in-process stub works quite smoothly in practice,
and is nearly indistinguishable from ptrace debugging.
In addition to meeting the no-tracing requirement, the
stub also enables tracepoints that are as fast and efficient
as compiled-in logging functions.

### 3 How Tracing Works

A tracepoint consists of a program location (function,
line number, etc), and a list of actions to be performed
when the program gets to that location. Most actions
simply say to collect a variable’s value and save it away,
but there is special syntax to ask to save all registers, all
function arguments, and so forth:

```bash
(gdb) trace prog.c:24
Tracepoint 1 at [...] 
(gdb) actions 1
```

Once all the tracepoints have been defined, the user uses
the `tstart` command to begin the tracing experiment
or tracing run. While tracing is active, whenever a trace-
point location is hit, the specified data will be saved to a
trace buffer, each such event producing one trace frame
(not to be confused with stack frames) of data. Tracing
can run concurrently with other debugging activities—
the program can stop at a breakpoint and then continue,
for instance.

Tracing can stop for any of several reasons:

- The user may order it with a `tstop` command.
- The trace buffer may become full.
- Each tracepoint can have a pass count, and when
  the tracepoint has been hit that many times, the ex-
  periment as a whole is terminated.

Once the trace run has completed, the user can use the
`tfind` command to select a trace frame to examine.
Trace frames can be found by number, or by tracepoint
number, or by address. The user may examine the col-
xlected data either by using the `tdump` command that
shows all of it, or by simply using normal `print`, `x`,
and other display commands; if the command asks for
data that was not recorded in the trace frame, GDB re-
ports an error.

The user may always start a new trace run by doing an-
other `tstart`. The trace commands may be modified
at any time, although the modifications do not take ef-
fect until the next `tstart`.

### 4 Tracepoints Become Breakpoints

The original implementation of tracepoints defined them
as their own distinct type of object, with their own com-
mands to create, delete, enable, disable, etc.

In practice, this has not worked so well. Tracepoints
have locations at which they are installed in the pro-
gram, they have actions to be performed when activated,
their addresses need to be updated when the program is changed; all things that breakpoints do as well. The net effect is that of a minor internal code fork, with the tracepoint version of the fork not getting improvements and fixes.

As non-breakpoints, tracepoints have missed out on several major improvements to GDB:

- Breakpoints can have multiple locations, in which a single source line may correspond to several program addresses. This is important for C++ template instantiations, since a single source line ends up at a different address for each instantiation.
- GDB's Machine Interface (MI) has not been extended to support tracepoints. Rather than provide a new set of MI commands, it would be better to extend the existing MI commands for breakpoints, and allow IDEs to extend their breakpoint handling.
- GDB's non-stop mode requires GDB have better knowledge of where tracepoints are located in order to deal with placing tracepoints and breakpoints at the same location. The existing implementation, which leaves tracepoint manages almost entirely to the remote agent, only suffices for all-stop debugging, where both tracepoints and breakpoints are removed when the target stops.

The actual implementation of the merge was relatively straightforward. We added a new breakpoint type `bp_tracepoint`, and added it to the various dispatches in breakpoint code.

As breakpoints, tracepoints have two main properties that distinguish them.

First, they have a list of actions to be performed when the tracepoint is hit. Although the action list is conceptually similar to the command list for regular breakpoints, actions are handled quite differently, and so we have kept them separate now. (For example, since actions are downloaded to the target and only executed there, they cannot refer to GDB's convenience variables.)

Second, and most importantly, tracepoints are downloaded to the target and GDB does nothing further with them. So execution control may assume that any stops are not due to tracepoints. This matters if a tracepoint and breakpoint have been set at the same address.

5 Tracepoint Action Improvements

For many cases, it is sufficient for a tracepoint action to record a few registers, and a few blocks of memory. To take the example of a statically allocated global buffer of 100 bytes, GDB simply compiles the action into a request to save 100 bytes of memory starting at the address of the buffer.

Many other cases are not so straightforward. If the buffer is dynamically allocated, then `collect buf` just collects the four or eight bytes of pointer to the buffer, which is going to be disappointing when the user says `print buf[3]`. To record that third byte in the buffer, we would say `collect buf[3]`, which means that the tracepoint handling code needs to be instructed to collect the value of the pointer, then add three, then dereference that to get the byte. This all has to be done in the target when the tracepoint is hit, with no help from GDB.

GDB does this by compiling expressions into a bytecode sequence that we call an agent expression, then having a simple bytecode interpreter on the target. In addition to its usual evaluation behavior, the interpreter takes note of any memory and register accesses, and records those as blocks of data in the trace buffer.

To see how this works, we have a handy maintenance command that runs the compiler and displays the bytecode sequence. (It needs registers, so the program has to be running, but you don't have to have a working tracepoint setup.)

```
(gdb) maint agent globarr[i+21]
0 const64 139937304 // addr of short array
9 reg 5 // base reg for local i
12 const8 240
14 ext 8
16 add
17 trace_quick 4 // collect 4 bytes at
19 ref32 // get value of i
20 ext 32
22 const8 21
24 add // compute i+21
25 ext 32
27 const8 2
29 mul // index into short
30 add // address of array
31 zero_ext 32
```
The byte code compiler in GDB works well, but since it is only used by tracepoints, it has not seen much use, and we have been filling in some omissions. We added logical and/or, conditional expressions, and comparisons. Also, C++ has not been much used with tracepoints, and we needed to add basic constructs such as this and reference types that were missing.

DWARF location expressions[6] are frequently used for local variables these days, and so we have also been adding to bytecode compilation for variables. This turns out to be a difficult problem, and is not yet fully handled; DWARF location expressions may themselves be arbitrarily-complicated bytecode sequences, and so we effectively need a general translator of location expressions into agent expressions.

### 6 Non-stop Debugging Versus Tracepoints

GDB’s original tracepoint scheme was designed for all-stop debugging, as that was the only debugging mode GDB provided at the time. In this mode, GDB would insert its breakpoints and then command the remote agent to continue. The remote agent would insert additional traps at each tracepoint location and continue the program. When a breakpoint was hit, the remote agent would check whether it matched one of the tracepoint locations, and if so it would process the tracepoint and then quietly continue, without saying anything to GDB. Otherwise it would remove all the tracepoint traps and report the breakpoint to GDB. The agent’s method of continuation from a tracepoint closely mimicks GDB’s behavior – the agent replaces the just-hit tracepoint trap with the original instruction, it single steps that thread, reinserts the tracepoint trap, and then resumes execution of all threads.

With non-stop debugging, continuing from a tracepoint becomes more complicated. Since multiple threads may be converging on a single tracepoint, the agent cannot remove the tracepoint trap, even momentarily, in order to single step past the tracepoint location. Instead, it must perform the single step at another location, by copying and adjusting the instruction underneath the tracepoint. Once again the agent mimicks GDB, in this case doing what GDB calls displaced stepping, and once again it happens entirely on the target.

One additional complication we discovered is what should happen with a breakpoint and a tracepoint are set at the same location. Because breakpoints and tracepoints must be inserted permanently, and GDB is unaware of the mechanism by which tracepoints are inserted, GDB’s explicit read-modify-write scheme of breakpoint insertion can lead to incorrect behaviour. It becomes necessary for the remote agent to manage breakpoints and support the z0 and z0 serial protocol messages. Then it is able to correctly deal with coincident breakpoints.

If a breakpoint and tracepoint are coincident, what should happen to a thread that hits them? Conceptually the breakpoint is between the previous instruction and the instruction underneath the breakpoint – the user sees all previous instructions executed when the breakpoint is hit, and when continued she sees all subsequent ones execute, including the one at the breakpointed location.

The tracepoint action needs to be done just before the breakpoint action, for pragmatic reasons. So conceptually, the tracepoint is just before the breakpoint. The reason for this ordering is because the alternative, of having the tracepoint just after the breakpoint is impossible to implement. GDB will single step that instruction out-of-place, and the remote agent will not know that any particular single step is the one for the breakpointed instruction (there could be many pending single steps in a multi-threaded program). Therefore, the remote agent must execute the tracepoint action, before reporting a coincident breakpoint to GDB.

### 7 Conditional Tracepoints

GDB supports conditional breakpoints and watchpoints by allowing the user to tack on an if and an expression to the command:

```
(gdb) break Foo if arg == 7
Breakpoint 1 at [...] 
```

In this case, the breakpoint on Foo will trigger only if variable arg is 7. In actuality, the breakpoint triggers every time and causes the program to stop, at which point GDB evaluates the condition and if true, leaves
the program stopped and reports it to the user, otherwise silently continuing the program.

It is useful to have a similar behavior for tracepoints. For instance, a particular piece of code may be frequently executed but only fail when some known but rare situation comes up. Without a conditional on the tracepoint, the trace buffer could very well fill up and stop the tracing run before the rare situation comes up.

We implemented conditional tracepoints, which simply use the breakpoint syntax:

```
(gdb) trace Foo if arg == 7
Tracepoint 1 at [...]  
(gdb) actions 1
>collect ptr, *ptr
>end
(gdb)
```

Unlike conditional breakpoints, a tracepoint condition must be evaluated directly on the target. Fortunately, we already have an expression evaluator, in the form of the bytecode engine described in Section 5. This required a handful of modifications to be useful for condition evaluation; we do not need to record anything for instance. If the condition is false, then the debug thread simply resumes the program on its own.

To see how conditional expressions turn into bytecode, we added a new maintenance command `maint agent-cond` that is like `maint agent`, but shows the bytecodes for the expression compiled as a conditional:

```
(gdb) maint agent globfoo < 45  
0  const64 139937272  
9  trace_quick 8  
11  ref64  
12  ext 64  
14  const8 45  
16  less_signed  
17  pop  
18  end
(gdb) maint agent-cond globfoo < 45  
0  const64 139937272  
9  ref64  
10  ext 64  
12  const8 45
```

The two key changes are that `globfoo` is simply gotten from memory and not saved into the trace buffer using `trace_quick`, and that the conditional does not pop from the stack. This example also shows us how the bytecode sequences can start to impose a performance burden, if used with a conditional tracepoint that is expected to be tested frequently, but rarely resulting in collection.

8 Trace State Variables

Conditionals that can test only program state have limitations. For instance, a traced location may only provide useful information after about the 50th time it is encountered, and so the user only wants to start recording then. To support that, we implement trace state variables. These are user-specified global variables that are maintained and manipulated on the target. They may be tested and manipulated in tracepoint conditions, and recorded in the trace buffer.

```
(gdb) tvariable $cnt  
Trace state variable $cnt [...]  
(gdb) trace prog.c:45  
Tracepoint 1 at [...]  
(gdb) cond 1 (($cnt=$cnt+1) > 50)  
(gdb) actions 1  
Enter actions [...]  
>collect $regs, $cnt  
>end
(gdb)
```

Trace state variables may be thought of as analogous to GDB’s convenience variables, but managed by the target rather than the host. Since GDB’s type and symbol facilities are not available on the target, we limit trace state variables to always being 64-bit signed integers.

We implemented one special trace state variable. The value of `$trace_timestamp` is taken from the system clock each time it is read, with a system-specific meaning.
9 Fast Tracepoints

So far, we’ve described tracepoints as a special kind of breakpoint, maintained by the target agent, but still implemented by inserting a trap instruction in the code stream. Although the tracepoint architecture removes several round-trip communications between GDB and the remote agent, it still involves kernel entry and exit, signal handling, and for a ptrace-style debug agent (such as gdbserver), context switches. On a fast x86 system this takes on the order of 500\(\mu\)S, and that can be too much of a time penalty for some applications.

In order to reduce tracepoint overhead, we need to avoid signals and traps altogether, executing the tracepoint code in the context of the traced thread. We do this by patching jump instructions into the executable. These jump to per-tracepoint trampolines. The trampolines must save some state and initialize for their distinct tracepoint before calling the tracepoint processing function.

9.1 User Interface

While this approach cuts out considerable overhead, it does have some major flaws. With the x86 ISA for example, jump instructions are not the smallest instruction. As we cannot, in general, know whether any particular instruction is the destination of a branch, we cannot overwrite multiple instructions with a single jump. Therefore, only locations with a suitably long instructions can be candidates for this kind of tracepoint. We resolve the conundrum by taking a page from hardware breakpoints, and introducing the fast tracepoint as a special type of tracepoint that the user must ask for explicitly with the \texttt{ftrace} command, for which GDB may respond that it cannot honor the request.

\begin{verbatim}
(gdb) ftrace 45
Instruction at 0x805e02c is \ 
only 2 bytes long, need at \ 
least 5 bytes
(gdb) ftrace 46
Fast tracepoint 1 at [...]
(gdb)
\end{verbatim}

9.2 Implementation of Fast Tracepoints

As with regular tracepoints, the instruction at the traced location must be executed out of line. In this case it is fixed up when the tracepoint is inserted, and placed at the end of the tracepoint’s trampoline, followed by a jump back to just after the original location (with suitable alterations for control flow instructions).

Inserting multi-byte instructions into a running program is particularly awkward on x86 hardware because of the variable length encoding. There is no guarantee that the instruction being changed will be naturally aligned – it could easily straddle a cache or page boundary. Thus it is not possible to atomically overwrite the instruction directly from a user process. Because insertion (and removal) occur in a non-stop multi-processor environment, there is always the possibility that some other thread is actively fetching or executing the instruction being executed. There are a number of race conditions, and multi-core processor errata that can lead to undefined system behaviour. Note that these errata do not apply to the single-byte breakpoint instruction. That can be inserted without problem, so breakpoints may be freely inserted in non-stop mode. Because of the atomicity and pipeline flushing requirements of the x86 architecture, a kernel module is required. This module is part of the application, and accessed through the API mentioned previously. Other architectures with fixed length instructions may be easier to work with.

Another difficulty with fast tracepoints is that the hit and collection processing happens within the context of each thread, rather than being done by the debug thread. The debug thread effectively serializes access to shared resources, most importantly the trace buffer itself—if several threads hit a tracepoint trap, they each wait in their signal handler until the debug thread works through its queue of threads needing attention. So if there is more than one tracepoint active, and one of them is a fast tracepoint, then the trace buffer must be locked with a mutex. This in turn introduces a possible deadlock, since the thread could receive a signal while it is processing its fast tracepoint, and then the signal handling code has a tracepoint installed in it (not unlikely, since tracepoints are a useful way to monitor for unusual signals).

9.3 Performance

Early results from fast tracepoints bear out the promise. A tracepoint with a conditional that evaluates to false should be the fastest of all paths, since they are likely to be inserted in critical code paths, with the conditional set to activate only on rare cases. On a 2+ GHz x86,
a conditional testing a single global can take under 300 nanoseconds. This is with very little tuning, and we expect to improve the numbers considerably.

10 Disconnected Tracing

One of the special advantages of tracepoints is that they can remain installed in a program for long periods of time, such as in a multi-day burn-in test, or in an attempt to catch a sporadic bug. However, the current design requires the GDB to remain connected to the target the entire time, and if the connection is broken due to a network failure, the collected trace data is effectively lost.

So we are introducing machinery that will allow GDB to disconnect from the target, while leaving the tracing run active. When disconnecting, GDB will remove all breakpoints and watchpoints, but if tracing is active, it will ask the user whether leave the tracepoints installed.

Upon reconnection, the target will inform GDB of the state of tracing; it might still be active, or might have stopped due to trace buffer fullness or passcounts being exceeded. It will also inform GDB of what tracepoints it knows about. This is necessary because there is no guarantee that the new GDB has the same set of tracepoints, or even if it does, the numbering scheme might be different because the user defined other types of breakpoints in the meantime. GDB will match up what tracepoints it can, and then inform the target of new numbers assigned to each tracepoint.

11 Tracing from a File

Extending from the disconnected tracing idea, the target system may have crashed, been shut down, or otherwise no longer be available. To deal with such cases, the application may have the ability to dump the trace buffer into a file.

Conceptually, this should be simple; define something like a `target tfile` command and write target operations to read requested registers and memory blocks from the file. But this won’t work, because the old tracepoint code does not go through GDB’s target vector at all! It actually directly composes packets in the remote protocol, and directly interprets the results.

So the key work here is simply to refactor the tracepoint code, and introduce a number of new target operations, basically one per type of tracepoint packet, about ten in all. The remote protocol implementation (remote.c) then does all the tracepoint packet encoding and decoding.

The target vector for trace files is fairly simple, in many ways reminiscent of corefile support. Since there is no running target in the picture, the necessary operations amount to just trace frame selection, register fetching, and memory reading.

12 Future Directions

Our few months of work on tracepoints has uncovered many opportunities for further improvements.

Merging of tracepoint actions and breakpoint command lists would reduce code duplication.

The bytecode compiler needs to be taught about the full range of DWARF location expressions.

The compiler could be made to generate more efficient agent expressions, which will reduce tracing’s impact on code performance still further.

It would be useful to have an option for a circular trace buffer, so that a too-rapidly-filling buffer doesn’t force the tracing run to terminate before reaching the most interesting parts of the program.

The trace state variable machinery should be generalized so that the target could inform GDB of the variables that it has available.

Finally, it would be useful to have a free version of target-side code suitable for use by applications. At the moment we have a chicken-and-egg situation, in that prospective users can only decide if they want to use tracepoints by putting considerable effort into writing custom target-side code. Even a low-performance version could be useful for evaluation, as well as enabling more thorough testsuite coverage.

13 Acknowledgements

We thank Ericsson for sponsoring this work. Dominique Toupin <dominique.toupin@ericsson.com> would like
to hear from people interested in leveraging this work for other functionality or to collaborate on additional GDB improvements.

Many thanks to Nathan Sidwell for assistance and advice during all parts of this project, and to Pedro Alves, who came to the rescue on multiple occasions.

References


Design of Graphite and the Polyhedral Compilation Package

Jan Sjödin  Sebastian Pop  Harsha Jagasia
Open Source Compiler Engineering, Advanced Micro Devices, Inc., Austin, Texas, USA
jan.sjodin@amd.com, sebastian.pop@amd.com, harsha.jagasia@amd.com

Tobias Grosser
University of Passau, Passau, Germany
grosser@fim.uni-passau.de

Antoniu Pop
MINES ParisTech, Centre de Recherche en Informatique, Mathématiques et Systèmes, Paris, France
antoniu.pop@mines-paristech.fr

Abstract

Graphite is the loop transformation framework that was introduced in GCC 4.4. This paper gives a detailed description of the design and future directions of this infrastructure. Graphite uses the polyhedral model as the internal representation (GPOLY). The plan is to create a polyhedral compilation package (PCP) that will provide loop optimization and analysis capabilities to GCC. This package will be separated from GIMPLE via an interface language that is restricted to express only what GPOLY can represent. The interface language is a set of data structures that encodes the control flow and memory accesses of a code region. A syntax for the language is also defined to facilitate debugging and testing.

1 Introduction

The polyhedral compilation package (PCP) is an optimization package that uses the polyhedral model as the internal representation to perform program analysis and transformations. Our goal is to define an optimization framework with clear interfaces to simplify testing and integration with GCC.

The polyhedral model can represent structured code containing sequences, linear conditions, well behaved loops, and affine memory accesses. The compilation unit is a static control part (SCoP), which does not have any side effects and all data accesses are statically determined to be linear. Array subscripts are limited to affine expressions of induction variables and constants: this restricts the data dependences to be regular, such that the data flow can be represented by unions of convex polyhedra. Scalar identifiers defined outside a SCoP are called parameters. Parameters cannot be modified in a SCoP. Parameters and arrays that are read inside a SCoP are inputs. The output of a SCoP are the arrays that have been modified and that are used after the SCoP.

In this paper, we will discuss the components of PCP and how they interact. Figure 1 shows an overview of PCP. GIMPLE is translated to the PCP language, which is in turn translated to the polyhedral representation GPOLY. The PCP optimizers, guided by a set of heuristics, exclusively work on the polyhedral representation to transform the code. These heuristics are based on information about the architecture, which must be provided by GCC in the form of a machine description.

To integrate PCP with GCC, there are four interfaces to consider:

- Language interface - defines a small imperative language used to represent a compilation unit.
- Polyhedral library interface - must be implemented to provide basic operations on polyhedra. Several polyhedral libraries exist and it is desirable to be able to use different libraries.
- Machine description interface - specifies the system the code should be optimized for.
- Transformation interface - allows GCC to specify specific transformations.
Finally, there are two more aspects of the framework: PCP constructs can encode auxiliary information, and PCP has an infrastructure for testing.

2 Language interface

The PCP language\(^1\) hides the internal representation (GPOLY) such that it can evolve without breaking backward compatibility with the translation to and from GIMPLE. The PCP interface language is restricted to express only what the polyhedral model can represent. PCP is a simple imperative language that only expresses communication between statements through array accesses and does not specify computations. Therefore the number of types and control flow constructs are fewer than in a general purpose programming language. Computations are encoded as “black boxes,” or user statements, that are parameterized with the reads and writes to arrays. The control flow constructs are structured loops and conditionals. PCP separates the identification of the structured regions of a GIMPLE program from the translation to GPOLY and clearly defines the information passed between GCC and GPOLY. The language constructs are relatively close to GIMPLE, which means the translation becomes fairly straightforward once a region of code has been identified. In addition to the data structures to represent the language, we have also defined a syntax to parse and emit PCP code.

In this section we describe the PCP language. The examples use the syntax that has been defined in the language specification\(^2\). The reason for having a textual language interface is to simplify testing and debugging. If there is no simple way to read and understand a piece of code, the debugging becomes a lot harder.

The syntax for the external language should be easy to read and write by humans and should not contain ambiguities. The expressiveness of the external language must not only be able to express all legal constructs, but also allow illegal constructs for negative tests.

Annotations and tests can be encoded in the language through optional arguments. Optional arguments encode extra information that is not needed to express the meaning of the program, but that is needed for other reasons. By specifying a standard syntax to allow parsing optional arguments, the parser can provide an AST for them. To eliminate ambiguities, such as operator precedence, and allow for a simple syntax for annotations, we use a functional (prefix) form for all language components.

2.1 Types

There is only one scalar type: arbitrary precision integer\(^3\) is used for array indexing, loop bounds, and linear conditions. The type is implicit and there is no syntax.

---

\(^1\)We refer to both the package and the language as PCP.
\(^2\)http://gcc.gnu.org/wiki/PCP
\(^3\)http://gmplib.org/
for it. The only types that must be specified are array types. Arrays types are defined by a list of constants or parameters that define the size of each dimension. If the list is empty, the type stands for a scalar type, for example:

```c
// Types:
myType <- array(10, 10)
myScalarType <- array()
```

### 2.2 Expressions

An expression is a linear combination of constants, parameters, and loop induction variables. Parameters are declared as inputs to a SCoP and never written inside the SCoP.

```c
// Parameter example:
myParameter <- parameter()
```

```c
// Expression example:
+(*(4, N), *(2, i), *(4, 4))
```

### 2.3 Array accesses

`def` and `use` define memory writes and reads. Each `def/use` takes a base array and a list of linear expression subscripts. A `maydef` encodes a possible write of a memory location, which may be used if there is control flow inside a user statement:

```c
// Array access example:
use(A, i, j)
def(B, +(i, j), k)
maydef(C, i)
```

### 2.4 Statements

Statements are the constructs that modify the machine state, either control flow or memory. User statements define computations that read and write arrays, but have no other side effects. The user statement consists of a unique name. The arguments to a statement completely define the memory operations done by the statement. The order of the arguments is maintained throughout the compilation. The access functions of `uses` and `defs` may be rewritten during the PCP transformations:

```c
// User statement example:
mystmt(def(B, i, j), use(A, -(i,1), -(j,1)))
```

The `copy` statement copies data from source to destination. The `copy` statement is a separate construct from the user statement, which allows PCP to introduce these non-computational memory operations. This construct may be used for “fan-out” communication patterns, as in array privatization.

```c
// Copy statement example:
copy(def(B, i, j), use(A, j, i))
```

The `guard` statement executes the body if the condition evaluates to true. There are two kinds of comparison operators: `eq` (equality) and `ge` (greater than or equal)

```c
// Guard example:
guard(eq(l, N))
{
  // Body
}
```

The `loop` statement takes four arguments. First a variable declaration for the induction variable. Second, an expression that defines the initial value of the induction variables. Third, a boolean expression that determines when the loop exits. Fourth, the stride (increment) of the induction variable after each iteration. The `loop` implicitly defines the induction variable. The induction variable can only be accessed inside the `loop` body.

```c
// Loop example:
loop(i <- iv(), 1, ge(N, i), 1)
{
  // Body
}
```

### 2.5 SCoPs

A SCoP is the compilation unit. It has a set of inputs and outputs. The inputs are scalar values (parameters), which are invariant in the SCoP, and arrays, which can be modified. Outputs are arrays that have been modified and will be used after the SCoP.

```c
// SCoP example:
scop(inputs(B, C), outputs(A), parameters(N))
{
  // Scop body
}
```
2.6 A complete example

Below is a small fragment of C code. Assume that the arrays A, B, and C have type double[1000][1000], and that N is a parameter.

```c
for (int i = 0; i < N; i++)
    { 
        A[i][0] = 0; 
        for (int j = 1; j < 100; j++)
    } 
```

The C code corresponds to the following PCP code:

```pcp
N <- parameter()
arrayType <- array(1000, 1000)
A <- variable(arrayType)
B <- variable(arrayType)
C <- variable(arrayType)
acop(inputs(B, C), outputs(A), parameters(N))
{ 
    loop(i <- iv(), 0, ge(N, i), 1) 
    { 
        stmt0(def(A, i, 0))
        loop(j <- iv(), 1, ge(100, j), 1) 
        { 
            // userStmt maps to the add and assignment
            stmt2(def(A, i, j),
                  use(A, i, -(j, 1)),
                  use(B, j, i),
                  use(C, -(j, 1), -(i, 1)))
        }
    }
}
```

2.7 Annotations

Annotations are used to represent auxiliary information that is needed for the compilation process. These can be added to any object in the language. Annotations should be handled by a generic framework, which will allow a compiler to track the information as the code is transformed. During the code generation, the annotations are added to the generated AST. Figure 2 shows the communication among the different components.

Annotations consist of a tag and a list of annotation arguments. An annotation argument can be a scalar value, an identifier, a string, or an annotation.

```c
// Annotation example:
A <- variable(array()) | myannotation() 
```

2.8 Test framework

The test infrastructure, as illustrated in Figure 3, takes text files containing PCP code as input. The input is parsed and dispatched to different components to perform the tests. Tests are specified either using annotations in the code or as a flags to the tester. For example, assume that the tester contains a test that checks if two statements in a loop can be distributed. This kind of test would test the dependence analysis. Assume that the associated annotation with the test is called `distributable`. This is an example of how a test case could be specified:

```pcp
for(i <- iv(), 1, ge(N, i), 1, 1)
    | distributable(stmt1, stmt2))
    { 
        stmt1(def(A, i), use(B, i))
        stmt2(def(C, i), use(D, i))
    }
```

Another example would be a check for loop fusion:

```pcp
loop1 <- for(i <- iv(), 1, ge(N, i), 1, 1)
    { 
        stmt1(def(A, i), use(B, i))
    }
```

```pcp
loop2 <- for(j <- iv(), 1, ge(N, i), 1, 1)
    | fusable(loop1))
    { 
        stmt2(def(C, j), use(D, j))
    }
```

If a test fails, the file name and line number where the annotation occurred is reported along with any diagnostic why it failed.

It is undesirable to use C or FORTRAN source code for unit testing since GCC is unlikely to be capable of producing all possible test cases. In addition, the test cases become unreliable because any of the passes before PCP may change and therefore modify the input.
to PCP. Test cases can be automatically extracted from C/Fortran code by using the PCP emitter to dump the SCoPs that are identified by Graphite. The emitter can also be used during debugging to produce reduced test cases that later can be added to the test suite.

The kinds of tests that are needed are both syntactic and semantic. Syntactic tests use simple string compare to check against the expected output. Semantic checks can be done both statically by analysis or dynamically by using an interpreter to execute the code. Since the actual computation is not represented in PCP, the result of the execution is the trace of memory accesses. Execution tests would mostly be used to verify the correctness of a transform by interpreting the code before and after the transform and comparing the trace results.

3 Translation of GIMPLE to PCP and back

Translating a region of GIMPLE to PCP requires the following steps:

1. Identify single-entry/single-exit (SESE) control regions in the CFG that can be represented in PCP and, thus, in the polytope model. This includes analyzing the control flow, loop structures, and induction variables and checking that all expressions for loop bounds, if-conditions, and array indexes are linear. A SCoP is defined in a context and is composed of a set of statements.

2. Detect relations between the parameters.

3. Detect natural loops based on the CFG or on the SESE structured program tree.

4. Identify the GIMPLE statements that will map to user statements. The statements that compose the SCoP are also called black boxes. A black box is a SESE region of the SCoP that describes a calculation. As we saw in the previous sections defining the PCP language, the only part exposed to PCP are the data references contained in the black box. As the name suggests, the scalar computations contained in a black box are hidden. A black box can contain a large set of statements, function calls, or irregular control flow, as long as the black box does not have side effects that are escaping the memory definitions and uses. A black box can be defined to encapsulate a part of the program that should not be transformed by PCP. Therefore, for efficiency reasons, one may want to use this mechanism to turn a part of a PCP program into a black box whenever the complexity of the polyhedral code generation is too high. Currently a black box is a basic block.

5. Construct PCP code.

3.1 Translation of PCP to GPOLY

The translation of PCP to the polyhedral model requires computing the iteration domain and the schedule for each statement in a SCoP.

---

A canonicalization pass is used to transform all expressions to a uniform format that makes it easy to generate constraints. The polyhedral library interface defines a linear expression as a vector of coefficients in which the position determines the variable or parameter the coefficient is multiplied with. The length of the vectors must be identical for all linear expressions in a constraint. For example:

\[
\text{and}(\text{ge}(i, N), \text{or}(\text{eq}(j), 5), \text{ge}(j, N))
\]

is translated to:

\[
\text{or}((\text{and}(\text{ge}(\{0,1\}, \text{eq}(0,j), \text{eq}(\{1,N\}, 0), \text{eq}(0,j), \{0,1\}, 5), \{0,1\}), \text{ge}(\{0,1\}, \text{eq}(0,j), \text{eq}(\{1,N\}, 0), \text{eq}(0,j), \{0,1\}, 5))))
\]

The resulting constraint system consists of a union of two polyhedra shown in matrix form:

\[
\begin{pmatrix}
1 & -1 & 0 & 1 & 0 \\
0 & 0 & -1 & 0 & 5 \\
1 & -1 & 0 & 1 & 0 \\
0 & 1 & -1 & 0 & 5 \\
\end{pmatrix}
\]

The first column in the matrix encodes if the constraint is an equality (= 0) or inequality (>= 0).

Uses and defs in the user statements are translated into linear expressions of the polyhedral library. The canonicalization has transformed the subscripts so they can be traversed and the coefficients can be extracted easily, which makes the translation straightforward.

The schedule of a statement is the time at which the statement is executed. There are two components to the execution time of a statement: the static time is the order in which a statement is executed in the sequence that is defined by the PCP abstract syntax tree. To define the static schedule, we use a Dewey numbering of the PCP abstract syntax tree. The dynamic schedule is represented by the iteration domain. Producing the schedule for all the statements is done by a traversal of the PCP abstract syntax tree.

The iteration domain is extracted syntactically from the PCP loop and guard constructs. In a PCP abstract syntax tree, each statement is contained in a set of loops and guards. Each surrounding loop defines a dimension in the iteration domain of the statement. The iteration domain for a statement defines the boundaries for the available induction variables. The guards define extra constraints and relations on the induction variables.

Figure 4 shows an example translating GIMPLE code to GPAOLY via PCP. The nested loops in the GIMPLE code maps to two PCP loops. In basic block 1 there is an in-
tialization of A, which maps to stmt1 in PCP. Basic block 2 contains a computation consisting of four GIMPLE statements that map to stmt2. The translation from PCP to GPOLY builds the iteration domains and schedule for each statement. The domains are defined by the loop bounds. The schedule is created by traversing the PCP code. For stmt1 the schedule is \((0, i, 0)\), which means it is the first statement at the top level and the first statement inside the \(i\) loop. The schedule of stmt2 is \((0, i, 1, j, 0)\), which is the first statement at the top level, the second statement inside the \(i\) loop, and the first statement inside the \(j\) loop.

3.2 Translation of GPOLY to PCP

The translation from GPOLY back to an imperative program is done by CLooG, which takes the iteration domains and schedule and produces an AST containing loops and guards.

Translating the CLAST to PCP is simple since both languages have the same constructs. In addition, CLAST gives a mapping for each statement that maps old the induction variables to expressions using new induction variables. All expressions in PCP are rewritten using this mapping.

3.3 Translation of PCP to GIMPLE

Translating PCP to GIMPLE is done by traversing the PCP structure and building the GIMPLE loop and conditions top-down. When a loop is encountered, a new loop structure is created in GIMPLE with a new variable that is the corresponding variable to the PCP induction variable. Each PCP induction variable is mapped to a new GIMPLE variable. When a user statement is found, the array accesses are translated and replace the old accesses in the original GIMPLE code.

4 GPOLY interface

The polyhedral representation of a PCP program is based on the following data structures:

- iteration domains
- scattering polyhedra
- data references
- data dependences

All these data structures can be accessed in read-only mode. The GPOLY transformations interface creates new scattering polyhedra from the original scattering. The original scattering represents the identity transform. The legality check for the transformed scattering is performed based on the original scattering.

4.1 Black box

The black box \(B = (\text{domain}, \text{drs}, \text{scattering})\) is defined by the iteration domain \(\text{domain}\), a set of data references \(\text{drs}\), and the \(\text{scattering}\) polyhedra.

4.2 Iteration domains

Each black box has an iteration domain represented with a union of convex polyhedra of dimension \(d\), where \(d\) is the loop nesting depth where the black box occurs. The iteration domain describes the set of iterations on which the black box is executed. The iteration domain does not describe the order in which the iterations are executed. The execution order, or dynamic time, is defined by the scattering dimensions of the scattering polyhedra.

4.3 Scattering polyhedra

A transform in the polyhedral model is a function that maps, for each statement, the original dynamic and static time to a new execution order. These transformation functions are also called scattering polyhedra, and are used to define an execution order, which provides the constrains necessary to produce an imperative code back from the polyhedral representation. The scattering polyhedra are expressive enough to represent all the loop and code motion transforms that are allowed in the polyhedral representation. They are composed of the following dimensions\(^5\):

- \(\text{scattering dimensions}\) represent the loops to be generated,

\(^5\)In this paper we will always use the name of the dimensions, and we will not define a mapping order for the dimensions. The reader can find examples of CLooG scattering polyhedra on http://gcc.gnu.org/wiki/Graphite/Scattering_polyhedron. Additional information about scattering polyhedra can be found in the CLooG documentation http://www.bastoul.net/cloog/manual.php#SEC8.
• *original iteration domain* are the dimensions of the original loop nest,

• *parameter dimensions* correspond to the variable names used in the program that are not varying in the current SCoP; parameters can be considered as induction variables of loops around the SCoP, and

• *inhomogeneous term* or *constant dimension*.

The scattering dimensions are a function of the original iteration domain, of the parameters and of the inhomogeneous term.

### 4.4 Data references

A data reference $DR = (alias\, set, subscripts, type)$ is defined by the *alias* set of the data reference. Every alias set is mapped to an unique value. If the array is part of more than one alias set, every array cell is mapped to one point for every alias set the array is part of. Then, each dimension of *subscripts* represents a subscript of the data reference. Scalar values are handled like arrays of dimension 0. The *type* of a data reference can either be read, write, or may-write. Read means a data reference reads or may read any of the values marked in accesses. Write means a data reference must overwrite all the values marked in accesses. May-write means that the values marked in accesses can be, but do not need to be, overwritten.

### 4.5 Legality and heuristics

The transformation engine in PCP will determine if a given loop transformation is legal based on the information obtained from the dependence analyzer and check if the transformation is profitable based on the information obtained from the transformation heuristics. To check if a transformation is profitable, the transformation engine will model the transformation by modelling the individual operations and comparing them with the machine characteristics provided by GCC in the form of machine descriptions. Based on the cost estimate, the transformation engine will decide the code generation and optimization. In some cases, the transformation engine may not be able to accurately determine the cost of the transformation because the passes after the transformation engine, like CLooG, can make further decisions to manipulate the code and have better knowledge of generated code characteristics like code size. In that case, a second profitability check will be done during PCP code generation.

Each user statement has costs associated with it: an execution time estimate and code size estimate. The execution time estimate is needed to determine the scheduling. The code size estimate is used to avoid code explosion when duplicating code, which could result in poor i-cache locality. The loop optimizations primarily focus on memory reuse, vectorization, and parallelization. Therefore the machine description must contain information about the memory hierarchy, vector instructions, and the configuration of the parallel system (e.g., number of cores and processors) and the latencies for communication.

To generate vector code, PCP annotates loops that are vectorizable (as independent), which can optionally be translated by the compiler in vector code. The compiler can encourage generation of vectorizable loops by giving lower costs to independent inner loops.

### 4.6 Polyhedral transform interface

Some of the operations in the polyhedral model have been discussed\(^6\) in [1, 2]. These operations are basic transformations of the scattering functions of statements. A similar interface will be provided in GPOLY, but only applies transforms to the scattering polyhedra.

This polyhedral interface is internal to the PCP library and is not exposed outside the polyhedral framework. A classical loop transform interface can be used to annotate transforms on the PCP abstract syntax trees and can be used to direct the transformations performed by PCP.

### 5 Loop transformation interface

PCP exposes a classical loop transform interface that can be used to drive the transformations that PCP applies. The interface is based on annotations that are set on the PCP trees:

\[
\text{loop1 (\ldots | fuse (loop2))}
\]

Appends the code of loop1 to the end of loop2.

\(^6\)http://www.lri.fr/~girbal/site_wrapit/
6.1 Invariants

Invariants is an extension to define further restrictions on scalar values. The compiler may have information about parameters or induction variables (for example, the type of a variable in the original program may restrict the range). The extension is an annotation that can be attached to an object or an expression. For example:

```c
// N < 256
N <- parameter((invariant(ge(255, N)))
```

6.2 Reductions

The copy statement can be used to expand (duplicate) data, but there is currently no way to express compression (reduction) other than a regular user statement. The problem with using a user statement for a reduction is that it induces a loop-carried dependence that cannot be parallelized or transformed. For example:

```c
loop(i <- iv(), 0, ge(N, i), 1)
{
    userStmt(def(A), use(A), use(B, i))
}
```

The `use(A)` and `def(A)` encode the reduction. The dependencies between two successive iterations of the loop are fixing the evaluation order; it would be illegal to parallelize or to perform some loop transforms.

To solve this problem, we introduce a reduction statement that provides extra information about the associativity and commutativity of a binary reduction operation. The reduction statement takes as a first operand the destination, and the second and third operands are the sources. With this extension, the previous example would be written as:

```c
loop(i <- iv(), 0, ge(N, i), 1)
{
    reduction(def(A), use(A), use(B, i))
}
```

This would now allow the loop to be marked as parallel:

```c
loop(i <- iv(), 0, ge(N, i), 1 | parallel)
{
    reduction(def(A), use(A), use(B, i))
}
```
6.3 User statements accessing induction variables

Currently a user statement may only access arrays. However, some computations may use the induction variables, which means a user statement must be able to directly access an induction variable. For example:

```c
loop(i <- iv(), 0, ge(N, i), l)
{
    stmt(use(i))
}
```

The assumption with this construct is that the use of an induction variable does not contain any dependencies that PCP must consider.

6.4 While loops

In some cases, the iteration domain may not be known before a loop starts to execute. To handle this case, we must introduce while loops. A while loop takes two arguments, an induction variable and a scalar variable that represents both the predicate and side effect of updating p1 in every iteration. The proposed syntax would be:

```c
// While loop example
while(i <- iv(), p1)
{
    // Body
}
```

6.5 Range specification for data accesses

Since a user statement can represent a larger control flow structure such as loop, it is possible that each invocation can read or write more than a single element of an array. To represent this in PCP, we must be able to specify a range for a subscript that is read or written by a statement. One proposal for specifying a range would be:

```c
stmt(use(A, range(0, i), j))
```

6.6 The mayuse annotations

The PCP language only defines the data accesses necessary to define correct semantics for the polyhedral transforms: these are def, use, and maydef. A mayuse could be used as a hint for the optimizers to detect locality properties of statements.

7 Conclusion

This paper provides a detailed description of the design and future directions of the Graphite and PCP infrastructures. PCP provides a language and a transform interface to represent and optimize data communications through array operations. The expressiveness of the PCP language is that of the polyhedral model: PCP programs can be translated in the polyhedral model and back to their imperative PCP format. The benefits of the PCP infrastructure are modularity, ease of debugging, and testing of the polyhedral transforms and analyses.

The paper provides technical details of the translation of PCP to the polyhedral representation GPOLY and back. The GPOLY interface provides data structures for a classical polyhedral representation, together with a set of transformations operating on GPOLY. An imperative loop transform interface is defined as annotations on PCP constructs. Finally we discussed extensions of the PCP language to capture a larger set of programs, for providing more precise information to the data dependence analysis, and hints for the cost models.

References


Porting GCC to Exposed Pipeline VLIW Processors

Alexandru Turjan
ST-Ericsson
alex.turjan@stericsson.com

Dmitry Cheresiz
ST-Ericsson
dmitry.cheresiz@stericsson.com

Roel Trienekens
Delft University of Technology
roel.trienekens@gmail.com

Abstract

EVP and TriMedia are embedded application processors targeted at mobile communication and multimedia domains. Both architectures originate from Philips Semiconductors and are currently developed by ST-Ericsson and NXP Semiconductors, respectively. Both processors have a VLIW architecture with an exposed pipeline. Such architectures impose different requirements on a compiler than the majority of existing GCC targets, which are scalar or superscalar machines with interlocked pipelines. First, the exposed pipeline organization requires a compiler to schedule operations such that all data and resource hazards are avoided. Second, a compiler for a VLIW machine has to provide stronger capabilities for discovering and exposing the instruction level parallelism (ILP), as it cannot rely on the hardware ILP mechanisms employed in superscalar processors. We have ported GCC to EVP and TriMedia and provided extensions to support code generation for an exposed pipeline VLIW. To increase the amount of exploitable ILP, we have also enhanced the current GCC mechanisms such as loop unrolling and the alias analysis. The ports were benchmarked against the existing production compilers and encouraging results in terms of cycle counts and code size have been achieved.

1 Introduction

Embedded systems-on-chip (SoCs) provide signal or media processing functionality for the products like mobile phones, TVs and set-top boxes. Such an SoC consists of several components. The host processor executes the control tasks such as running embedded OS, providing user interface, etc. Typically, the host processor is a general purpose processor (GPP) based on an existing architecture, such as ARM or MIPS. It controls the rest of the system and dispatches the tasks to one or several application processors (APs) and, optionally, hardware accelerators. An application processor provides the core functionality of an SoC and runs the most compute intensive tasks. The Embedded Vector Processor (EVP) [1, 2] and the members of TriMedia processor family [3, 4] are examples of such processors. An application processor is required to provide very high performance/power ratio on a limited set of algorithms from a certain media or signal-processing domain. For example, for the inner receiver of the LTE standard, at which EVP is targeted, an application processor should provide 13 GOPS with a power budget of less than 400 mW.

The algorithms mapped on APs exhibit significant amounts parallelism which facilitate the achievement of performance/power targets. Data-level parallelism (DLP) is commonly exploited by means of vector instructions. For example, EVP uses dedicated 256-bit vector registers, while TriMedia provides vector operations on existing 32-bit scalar registers. We remark that contemporary GPPs take the same approach in exploiting the DLP, introducing short-vector instructions, e.g. Altivec and SSE ISA extensions. For exploiting the instruction-level parallelism (ILP), however, the approaches used in GPPs and in embedded APs differ significantly. Due to need for legacy code support, binary code compatibility is often a must for a GPP. Therefore, such a processor accepts sequential code and exploits the ILP by means of hardware mechanisms which discover the independent instructions in the program flow and issue/execute them in parallel. Compiler for such a processor plays a limited role in exposing the ILP, leaving the main responsibility to the hardware.

This approach would be too costly for an embedded pro-
Currently, EVP and TriMedia are supported by tool chains based on proprietary compilers [5, 6]. However, rapid development of GCC and introduction of new features such as auto-vectorization motivated us to investigate its capabilities with respect to these processors. This has lead to the setup of two study projects in which we ported GCC to EVP and TriMedia and investigated its advantages and limitations, particularly with respect to exposed VLIW architecture support. In this paper we present the experience gained during these projects, the techniques we employed to target GCC, and some of the achieved results. The goal of the projects has been to generate correct and efficient VLIW code using GCC, and, desirably, to achieve the level of performance comparable with our existing production compilers.

The main steps in VLIW compilation are sequential code generation and scheduling. The first step consists of converting the input code into a sequence of operations supported by the target. It corresponds to the usual compilation flow for scalar or superscalar processors and is performed in the same way. During the second step, the individual operations are packed together into VLIW instructions such that the data dependencies and resource constraints are satisfied. This often requires insertion of naps within and between VLIW instructions. In Section 2 we present the EVP and TriMedia VLIW architectures and describe the scheduling task for them in more detail. Section 3 presents how we adapted GCC to schedule correct VLIW code, presenting the use of the GCC’s internal DFA-based scheduler for basic blocks, and two custom-made algorithms that we have developed: the inter-basic block scheduling pass guaranteeing that the scheduling constraints across the basic block boundaries are satisfied, and the resource-aware branch delay slot scheduler.

To achieve performance on a VLIW target, it is crucial that sufficient parallelism is exposed to the scheduling pass. In Section 4 we present several common VLIW compilation techniques used for this purpose and our GCC implementation of them. The techniques include loop unrolling precisely controlled by the unroll pragma and the address-based alias analysis. We remark that these techniques are applicable and can be beneficial for any processor that exploits ILP. In Section 5 we present the results which were achieved by implementing the aforementioned methods and report the performance of our GCC ports. Section 6 summarizes our conclusions.

2 Architectural Overview

The instruction set architecture (ISA) defines the compiler’s (programmer’s) view of a processor and its resources. The level of detail with which a processor is represented in an ISA can vary considerably. For the ISAs commonly used in general-purpose processors, this level is rather low, describing the registers, the memory and its addressing modes, and the instructions. The processor pipeline details are hidden from the compiler, which facilitates binary compatibility of different processors implementing the same ISA. When, for example, the pipeline depth, the instruction latencies, or the number of instructions that can be issued in parallel change in a new generation of a processor, old binaries can be executed without recompilation. This motivates the use of such ISAs in GPPs. We remark, however, that in this case the processor is responsible for the correct execution of the code. To achieve this, the processor contains a number of interlocks, which control the flow of instructions and data through the processor pipeline. An interlock is a piece of hardware which detects if an instruction at given pipeline stage can proceed to the next stage without causing data hazards and structural hazards [10] (also referred as resource conflicts). If this is not the case, the interlock hardware stalls (a part of) the processor pipeline till all hazards are resolved.

For VLIW processors in the embedded domain, binary compatibility is less of an issue, while the hardware complexity should be minimized. Therefore, a VLIW ISA employed there usually exposes more details of the processor pipeline organization to the compiler. Typically, the number of operations which can be encoded in a single VLIW instruction (referred as the number of issue slots) and the latencies of the operations are visible to the compiler. The responsibility of its scheduling pass is to bundle together the operations that can be executed in parallel without causing data or pipeline haz-
ards. The unoccupied issues slots in a VLIW instruction are padded with \textit{nops}. The scheduler attempts to place the data-dependent operations in different instructions at sufficient distance from each other to avoid the data hazards. Similarly, it tries to schedule at sufficient distance the operations that occupy the same pipeline resource in order to avoid resource hazards. A VLIW machine often lacks (most of the) interlocks. The effect of a hardware interlock on the flow of instructions through the pipeline is similar to an insertion of a \textit{nop} instruction in the program code. In case interlocks are absent, nop instructions should be explicitly inserted in the code by the compiler or the programmer. For a VLIW machine with a non-interlocked pipeline, the scheduling pass is responsible for inserting the nops between the VLIW instructions, so that all the hazards are avoided.

EVP is an 11-slot VLIW with a 9-stage non-interlocked pipeline. A single EVP VLIW instruction can issue in parallel up to 5 scalar operations and up to 6 vector operations. The individual operations can be predicated. We remark that in order to avoid excessive number of ports on the general-purpose register file (RF), EVP contains separate \textit{pointer} (\texttt{ptr}) and \textit{offset} (\texttt{ofs}) register files for address computations, and the \textit{predicate} RF. Vector operations typically work on $16 \times 256$ bit general-purpose vector registers from the \texttt{vr} RF. Due to register file port considerations mentioned above, the vector registers meant for specific type of vectors are contained in separate vector RFs, such as \textit{vector shuffle pattern} RF \texttt{vsp}, \textit{vector mask} RF \texttt{vm}, and several others. EVP is targeted at baseband signal processing, and one of its salient characteristics is the combination of the VLIW and vector processing with DSP features such as zero-overhead loops and circular addressing modes. For further details on EVP architecture, the reader is referred to [1].

TriMedia is a classic VLIW architecture which has been successfully implemented in several application processors targeted at multimedia domain. Its baseline instruction set consists of RISC-like operations working on a large register file consisting of $128 \times 32$ bit general-purpose registers (GPRs). These registers are used for arithmetic and memory operations, as well as for address calculations and predication. In addition to the typical RISC operations, TriMedia contains a rich set of special-purpose operations for media processing (referred as \textit{customops}). Typically, they operate on GPRs seeing them as vector registers containing short vectors consisting of four 8-bit or two 16-bit elements. Recent generations provide customops operating on two concatenated GPRs, thereby increasing the vector length. Most of TriMedia processors are non-interlocked 5-slot VLIW [4].

3 Scheduling for an Exposed Pipeline VLIW Using GCC

GCC is primarily developed for processors with interlocked pipelines. Therefore, by default, GCC produces sequential assembly code and does not perform packing of VLIW instructions and insertion of nops, except for the nops in the branch delay slots. One way to provide a GCC-based compiler for an exposed VLIW processor would be by reusing the scheduler of an existing compiler for the machine and passing GCC’s output through it. We have taken this approach for TriMedia, where GCC produces sequential code\footnote{This, essentially, is the code for a 1-slot interlocked machine with TriMedia ISA.} for TriMedia ISA, which is scheduled afterwards by the separate \textit{tm sched} scheduler [14].

Such an approach for EVP was not feasible because its scheduler was integrated in the proprietary CoSy-based compilation toolchain [5] and not available standalone. However, we were able to schedule correct and efficient VLIW code for EVP using the GCC framework by employing the internal GCC’s scheduler and providing some additional functionality, as presented in the remaining part of this section.

3.1 DFA-Based VLIW Scheduling of Basic Blocks

Interlocked processors do not require instruction scheduling. Appropriate scheduling, however, facilitates reduction in the number of runtime interlocks and thereby improves performance. For this reason, GCC includes an instruction scheduler, which is implemented in the \textit{haifa-sched} pass. The main algorithm performs top-down priority-based list scheduling on a basic block (BB) of RTL instructions and is implemented in the \texttt{schedule_block} function. At each cycle the algorithm attempts to schedule instructions which are \textit{ready}, i.e., the instructions of which the data dependencies have been satisfied. In case a nonzero number of instructions have been scheduled at the current cycle, the mode of the first one is set to \texttt{TImode}. If no instruction have
been scheduled, we insert in the code a new RTL instruction \texttt{(const\_int 0)} which represents a nop, and tag it with the TMode too. The nop insertion is done by the target hook \texttt{TARGET\_SCHED\_REORDER2}, which is called after an instruction is issued. It compares the current cycle with the cycle at which the previous instruction has been scheduled. If the cycles are not consecutive, it inserts an appropriate number of nops. Since the mode of all other instructions is VOIDmode, TMode tagged instructions represent the borders of VLIW instructions. In the final stage of the compiler we scan the instruction sequence using the TMode tags to emit assembly delimiters representing VLIW packing.

When the last predecessor of an instruction $A$ is scheduled, the algorithm guarantees that no data hazards will happen by sufficiently delaying the cycle when $A$ will become ready. The resource hazards are avoided when the algorithm attempts to schedule a ready instruction. It queries the \textit{Deterministic Finite Automaton (DFA) based pipeline hazard recognizer} [9], which determines if scheduling an instruction at the current cycle will cause a resource conflict. In such a case the instruction is queued for the number of cycles needed to resolve the conflict. Pipeline resource descriptions used by DFA can be also utilized to specify other scheduling constraints. For example, the EVP instruction format provides only a single opcode field for a 32-bit \textit{long} immediate. Therefore, only a single operation in a VLIW instruction can have such an immediate operand, while several \textit{short immediate} operands fitting in a narrower range are allowed. To satisfy this restriction, we specify a special pipeline resource representing the long immediate field in the opcode, and reserve it when scheduling an operation which requires such an immediate.

For EVP, due to a large number of issue slots and different instruction classes the generation time of a single DFA scheduling automaton became impractically large. In order to cut down the build time of the compiler the automaton was split into two separate automata: all functional units belonging to the scalar and address calculation parts of the processor are represented in the automaton \textit{scalar}, and all vector functional units – in the \textit{vector} automaton. This split resulted in two much smaller automata\footnote{The combined number of states and arcs in the two automata is roughly 37000 and 250000, respectively. For a unified automaton, these numbers would be at least a 1000 times larger.}. The majority of the instructions reserve resources from a single automaton. The only exception are certain vector operations that reserve resources from the \textit{vector} automaton, as well as the long immediate unit which belongs to the \textit{scalar} automaton.

### 3.2 Inter-Basic Block Scheduler

The scheduling algorithm described above is applied to all basic blocks in the program’s \textit{Control Flow Graph (CFG)}. For each block $B$, it guarantees that the generated schedule will satisfy the constraints imposed by the operations belonging to $B$. This, however, is not sufficient for correct code generation. Suppose basic block $A$ is an immediate predecessor of $B$ in the CFG. Consider an operation $op_j \in A$ which is scheduled, for example, in the last cycle of $A$. Let $op_j \in B$ be an operation which uses the result of $op_j$. To satisfy the true data dependency, $op_j$ should be scheduled not earlier than at cycle $\text{latency}(op_j) - 1$ (cycle counts in a schedule starts from zero). When scheduling $B$, the algorithm implemented in \texttt{schedule\_block} is not aware of cross-block dependencies and might assign $op_j$ to an earlier cycle, thereby creating an incorrect schedule. To fix such mistakes, we have implemented an additional \textit{inter-basic block scheduler}.

For each basic block $B$, this function iterates over all of its instructions and checks whether dependent instructions located in all of the predecessor BBs are scheduled at sufficient distance. To find data dependent instructions from two blocks $A$ and $B$, we make use of the GCC data structures which keep the live-in/live-out registers. Once two dependent instructions $op_i \in A$ and $op_j \in B$ are detected, the algorithm calls the backend-specific function \texttt{insn\_latency(\textit{op}_i, \textit{op}_j)} which returns the number of cycles that have to be executed to satisfy the dependency. In case the distance between the instructions is too small, we insert an appropriate number of extra nops at the top of $B$.

Similarly to dependencies, resource reservations related to the scheduling of an instruction in one basic block may impose additional scheduling constraints for instructions in the subsequent blocks. To account for such effects, we provided an additional DFA-based algorithm which guarantees that all the resource conflicts are avoided. A similar mechanism have been also employed in our custom-made branch delay slot scheduler which is described below. Both schedulers operate on RTL code which has been already scheduled in GCC...
sched2 pass, and take place in the customized reorg pass executed shortly before the final assembly emit pass

3.3 Branch Delay Slot Scheduling

The existing GCC Branch Delay Scheduler (BDS) takes into account only the program data-dependencies and ignores the resource constraints, assuming that they will be ensured by hardware interlocks. To produce correct VLIW code, we disabled the original BDS pass and, initially, filled all delay slots with nops. However, the number of delay slots in EVP is quite large (5 or 7) and the associated performance penalty was considerable (e.g. \(-20\%\) for EEMBC-telecom). This made evident that a BDS pass is crucial to achieve satisfactory performance. The existing GCC BDS is a sophisticated algorithm and making it aware of resource constraints would be a rather challenging task. Hence, we decided to implement a proprietary resource-aware BDS for EVP. The algorithm iteratively attempts to move the branch up within the block. It starts with the last instruction preceding the branch and attempts to move the branch above it. This is equivalent to moving the instruction into a delay slot below the branch. If the move was successful, the algorithm tries to move the branch one more cycle up, proceeding in this was as long as moving is possible and there are non-filled delay slots. Each of aforementioned moves should not violate existing dependencies and should not create resource conflicts. The first condition is satisfied automatically: suppose instruction \(y\) depends on instruction \(x\) and \(y\) is moved into branch delay slot. Essentially, it means that the branch is inserted between \(x\) and \(y\), thereby increasing the distance between them by \(1\) and satisfying the dependency.

Avoiding resource hazards requires more care. Suppose \(op_i\) is scheduled at cycle \(c_1\) and uses resource \(r\) at cycle \(m\) and \(op_j\) is scheduled at cycle \(c_2\) and uses resource \(r\) at cycle \(n\). Consequently, resource \(r\) is used by \(op_i\) at cycle \(c_1 + m\) and by \(op_j\) at cycle \(c_2 + n\). The GCC instruction scheduler ensures that \(c_1 + m \neq c_2 + n\). Suppose that \(op_j\) is moved into a delay slot. As a consequence resource \(r\) is used by \(op_j\) at cycle \(c_1 + n + 1\). In such case our BDS has to ensure that \(c_1 + m \neq c_2 + n + 1\). We solve this by making use of the GCC instruction scheduler; we artificially impose that \(op_i\) uses the resource \(r\) at cycle \(n - 1\) as well. For example in case of EVP the div operation which was using the register file write port at cycle 10, while the alu operations use it at cycle 3. Therefore, we impose that div uses the write port also at cycle 9. For an arbitrary processor resource utilization such solution may become too costly. However, for EVP we had to employ this technique only for a limited number of instructions. According to our benchmarking the change in the resource utilization of those instructions didn’t induce performance penalty.

3.4 Scheduling semantically equivalent operations

To achieve higher performance, EVP allows scheduling of some operations on different functional units. This allows several such operations to be scheduled in parallel in a single VLIW instruction. For example, moving data between two general purpose registers can be issued on two different functional units c_salu_1 or c_slsu_s1. These moves have different assembly syntax, move and move_slsu, and can be issued in parallel. Implementing this functionality using DFA caused the complications described below.

GCC emits an assembly mnemonic for an RTL instruction based solely on the set of operand constraints which it satisfies (recorded in GCC’s which_alternative variable) and is agnostic of the instruction’s DFA resource reservations. The two move instructions mentioned above have the same semantics and, therefore, their RTL templates and the operand constraints are identical. Hence, GCC has no means of differentiating them and emitting different mnemonics. To resolve this issue we have taken the following approach.

For an RTL instruction which can be issued on different functional units (with different mnemonics), during the sched2 pass we use the DFA to determine the functional unit on which the instruction was scheduled, and append to it an additional RTL pattern of the form: (clobber (match_operand N "const_int")), where the value \(N\) of the clobber operand is used to encode the information about the selected unit. This information stays attached to the instruction rttx till the final stage of compilation, allowing in this way that the proper assembly mnemonic is emitted. For example, the 16-bit move instruction can be issued on SALU and SLSU unit, and assembly generation is done as follows:
In order to query the DFA automaton about scheduling decisions and to record this information in the clobber rtx, we have implemented the evp_automaton_query function and tied it to the target hook TARGET_SCHED_DFA_NEW_CYCLE, which is called every time a new instruction is about to be scheduled. The main parameter of the function is ready rtx which represents the current instruction considered for scheduling. We illustrate the functionality of evp_automaton_query using the aforementioned 16-bit register move instruction as an example. In this case ready is RTL instruction of the form (set (reg:HI ri) (reg:HI rj)). First, we create two temporary RTL instructions, insn1 and insn2 which belong to c_salu_1 and c_slslu_sl classes, respectively. Then we attempt to schedule each of these instructions at the current DFA state, by calling the internal_state_transition() function. Suppose scheduling of insn2 was successful. This fact is memorized by assigning the variable key=33. Afterwards we trick the compiler into thinking that ready should be scheduled according to the c_slslu_sl pattern as shown below:

```c
int uid = INSN_UID(ready);
dfa_insn_codes[uid] = internal_dfa_insn_code(insn2);
```

The DFA scheduler uses the array dfa_insn_codes[] (indexed by the instruction number uid) in order to store for each instruction its instruction class (and, hence, its resource reservations). The array elements represent the internal DFA codes of instruction classes which can be obtained by calling internal_dfa_insn_code(). By assigning the dfa_insn_codes[] for ready as shown in the code fragment above, we essentially force the compiler to think that the resource reservations of ready are as of the c_slslu_sl class, and to schedule it correspondingly.

The final action performed by evp_automaton_query just before exit is the attachment of (clobber (const_int 33)) rtx to the original RTL pattern of ready. Upon exit from the function, the DFA scheduler will attempt to schedule ready. It will look up its instruction class in dfa_insn_codes, finding that it is c_slslu_sl, and will successfully schedule it.

We remark that in a case where all the alternatives contain resources from a single automaton, a different implementation of evp_automaton_query would be possible, based on the existing DFA facility which allows an instruction class to specify several scheduling alternatives using the OR construct (e.g. "c_salu_1 | c_slslu_sl"). Such an implementation, however, would lead to creation of a considerably larger DFA then in our method. Furthermore, in case the original DFA has been split into two or more automata (e.g., scalar and vector DFAs in case of EVP) and an instruction contains scheduling alternatives which belong to different automata, the DFA scheduler would not treat it correctly. Our implementation, however, can treat such scheduling constraints properly.

## 4 Increasing ILP Exposed to the Scheduler

Majority of existing high-performance CPUs supported by GCC are superscalar processors, in which hardware mechanisms are employed to expose and exploit ILP. For example, branch prediction and speculation allow the processor’s fetch and decode engines to run ahead of the execution and buffer decoded instructions from different basic blocks. In this way, the ILP across the basic block boundaries is exposed to the execution hardware which exploits it by issuing each cycle multiple instructions from the buffer, usually out-of-order, and guarantees that data dependencies and recourse constraints are respected. Effectively, it performs run-time scheduling. Hardware register renaming and dynamic memory disambiguation are often employed to remove false register and memory dependencies, thereby increasing the ILP and allowing more instructions to be issued in parallel.

For VLIW processors like EVP and TriMedia, the task of exposing and exploiting the ILP is shifted to the

---

4We have brought this issue to the attention of Vladimir Makarov, the developer and maintainer of DFA functionality. For the details we refer an interested reader to a corresponding discussion in the GCC mailing list.
scheduling pass of the compiler. Specific architecture features and compilation techniques are employed, allowing the scheduler to statically carry out the tasks which in superscalar processors are performed dynamically by the ILP hardware. For example, in order to remove the anti and output register dependencies, superscalar processors dynamically rename the compiler visible registers described in ISA to a larger set of hardware registers. To achieve similar effect in Tripedia, a much larger set of registers is provided directly in the ISA, and the compiler statically renames the registers that cause false dependencies.

The scheduler operates on a certain scheduling scope, also referred as scheduling unit. This scope is usually given in terms of basic blocks, and can range from a single BB to the complete CFG. As the ILP available in a single block is limited, superscalar processors employ branch prediction and speculation to discover the ILP across the block boundaries. To achieve a similar effect, scheduling for VLIW machines is performed on multiblock scheduling units. For example, a Tripedia scheduling unit is a decision tree (dtree), which is a CFG subgraph with the single entry and multiple exits [14].

The task of a VLIW scheduler is to assign to each operation $op$ in the scheduling unit an integer $c(op) \geq 0$, which denotes the order of the VLIW bundle to which it belongs. This integer is also the number of the cycle (counted from the beginning of scheduling unit execution) at which the operation will be issued. The generated parallel code should preserve the semantics of original program. To achieve this, the scheduler detects data and control dependencies between the operations and schedules them such that the dependencies are preserved.

The quality of the final schedule depends on two main factors: the amount of ILP present in the scheduling unit and the capability of the scheduling algorithm to extract and utilize this parallelism. The latter factor constitutes a complex subject for a standalone study which falls outside the scope of this paper. Therefore, the remaining part of this section is dedicated to a number of techniques which increase the amount of ILP and describes how they were supported in our GCC ports for EVP and Tripedia. Two approaches are commonly used to expose more ILP:

- **Scheduling scope increase** provides the scheduler with larger number of operations, and therefore increases the chance to find the independent ones, which can be executed in parallel.
- **Reducing dependencies** between operations increases the scheduling freedom thereby increasing the chance to schedule them in parallel.

The enhancement to GCC loop-unrolling which allows scheduling scope increase in a precisely controlled fashion is presented in Section 4.1. The scheduling scope in our port has been also increased by application of if-conversion and tail duplication. For these passes, however, we employed the existing GCC implementations, which have certain limitations. GCC tail duplication pass, for example, applies the transformation relying on internal compiler heuristics and does not provide direct control to a programmer. Development of such functionality would enhance this technique and constitutes an interesting subject for the future work. Section 4.2 describes the address-based alias analysis on the RTL which we implemented in order to improve the existing GCC memory disambiguation capabilities. Alias analysis allows static disambiguation of memory accesses thereby reducing number of false dependencies between them.

### 4.1 Controlled Loop Unrolling

Loop unrolling is a common code transformation which replicates the loop body several times. It creates a larger segment of non-loop code and, consequently, facilitates creation of a larger scheduling scope. Additionally, it decreases the number of updates of induction variables and the number of loop exit tests. GCC contains two unrolling phases: the first one works at the Gimple level and does total loop unrolling while the second one operates at the RTL level and does partial unrolling. These phases did not completely suit our needs and had the following limitations. The total loop unroll phase requires the iteration count to be a statically known constant, which is not always the case. Furthermore, the code size penalty resulting from the total unroll can be unacceptable. Partial unrolling is more suitable for our purposes. However, the corresponding GCC phase induces the unroll factor based on heuristics, and can be controlled by the programmer only indirectly by means of the following hooks: \texttt{PARAM\_MAX\_AVERAGE\_}
Despite using these facilities, we were not able to steer the compiler towards achieving the optimal unroll factor for all the cases and, consequently, observed significant performance penalties on certain benchmarks. A TriMedia or EVP programmer chooses the unroll factor very carefully. Insufficient unrolling does not expose enough parallelism. Excessive unrolling, on the other hand, leads to increased code size and creates too much register pressure, which results in spills and performance degradation. In TrimMedia, the selected unroll factor is communicated to the compiler by means of the unroll pragma of the following form:

```
#pragma TCS_unroll n
```

where `n` represents the unroll factor. This pragma is heavily utilized for optimization of the production code.

As the support for such precisely controlled unrolling was missing from GCC, we have added it in our back-end. Initially, we have considered the GCC facility for adding attributes which could have potentially being extended to support the unroll pragma. However, currently the attributes can be only attached to functions and not to the loops. Therefore, a different approach has been taken, as described below. Instead of an attribute, we attach to a loop a new special-purpose `unroll` RTL instruction which holds the unroll factor. During the RTL unroll phase, we analyze each loop and, when present, retrieve the associated unroll instruction. The unroll factor is extracted and applied to the loop, and the instruction is discarded.

The association between the pragma and the special RTL instruction is realized as follows. First, we add a new built-in function `__unroll pragma()`, which has a single integer parameter representing the unroll factor. Second, the `REGISTER_TARGET_PRAGMAS` hook is employed to introduce the new unroll pragma to the compiler. The `trimedia_unroll pragma()` function is tied to this hook and is called during parsing each time when the pragma is encountered in the source code. This function substitutes the pragma with a call to `__unroll pragma()`. Finally, during the RTL expansion, the call to the builtin is substituted with the unroll instruction RTL:

```c
{define_insn *customop_unroll pragma
 [(unspec_volatile:SI
   [(match_operand:SI 0 "immediate_operand" "I")
     | UNSPEC_unroll pragma)]
  **
  **
)
```

We remark that the instruction is declared as `unspec_volatile` in order to avoid it being moved away from the corresponding loop during the optimization passes.

### 4.2 Address-Based Alias Analysis on RTL

Alias analysis (AA) is a technique that allows to recognize if two pointers do not refer to the same address (i.e., alias). Stronger alias analysis allows to reduce the number of dependencies between memory operations in a scheduling unit. This increases amount of ILP that can be utilized and, potentially, leads to a shorter schedule. Strong AA is particularly important for making loop unrolling and software pipelining to be effective on a VLIW machine. In this techniques, scheduling scope consists of operations belonging to several loop iterations. Consequently, memory operations from different iterations will be present in the scope. If AA is weak, spurious dependencies will be created between the memory operations. These dependencies limit the scheduling freedom and the amount of cross-iteration ILP that can be utilized.

GCC provides AA support at both the GIMPLE and the RTL level. The Gimple AA has been introduced within the Tree SSA infrastructure, while the RTL AA is due to the old (before version 4.0) RTL-based infrastructure. The Gimple AA includes type-based analysis and points-to analysis. Type-based analysis makes use of the C language aliasing rules. It checks the pointer types of two memory accesses and, in case they are different, concludes that the accesses are disjoint.

Points-to analysis (or pointer analysis), is a technique that establishes to which variables or storage locations an arbitrary pointer points to. The variables or storage locations are united into sets, which afterwards are used to disambiguate arbitrary pointers. Using the code fragment below we illustrate the capabilities of the points-to analysis.

```c
int *p, *t;
int a[10], b[10], c[10];
if (d > 10)
  p = b;

Exceptions: 1) one may use a pointer or reference to a signed type to access an object of unsigned type, or vice versa, 2) one can use a pointer or reference with different const-ness or volatile-ness than the object, and 3) one can use a pointer of type char or unsigned char to access any object.
else
  p = c;
  t = a;
for (int i = 0; i < d; i++)
  *(t+i) = *(p+i);

Based on this analysis GCC correctly finds out that p points to the set \{b,c\} and t to the set \{a\}. As the sets are disjoint, GCC concludes that the two pointers do not alias. We remark that when points-to analysis is employed for an unrolled loop, its capabilities are limited. It will be able to disambiguate unrolled stores (i.e., \(*\{t+0\}, \*\{t+1\}...\)) from unrolled loads (i.e., \(*\{p+0\}, \*\{p+1\}...\)). However, it will not be able to disambiguate among different stores. As points-to analysis is performed at GIMPLE level, it can only be employed for the loop totally unrolled using the GCC unroll pass on the GIMPLE level.

In Section 4.1 we have presented why partial and precisely controlled loop unrolling on the RTL level is desirable for a VLIW machine. After implementing this functionality in our port, we have observed that the performance gains were limited. The reason for this is the weakness of existing AA on the RTL level. The current RTL AA is mostly type-based and therefore, cannot disambiguate the stores \(*\{t+0\}, \*\{t+1\}...\) from the loads \(*\{p+0\}, \*\{p+1\}...\). Consequently, although the loop gets unrolled, little or no cross iteration ILP is extracted. The purpose of the work presented in this section is to improve AA on RTL, so that the benefits of RTL loop unrolling for VLIW scheduling can be reaped.

Improving the RTL AA can be done in many ways. One option consists of improving the transfer of information between Gimple and RTL. To achieve this, one has to adapt the alias information model used by the two compiler representations: Gimple uses explicit representation in terms of points-to sets, while RTL is relies on a query-based disambiguation, i.e., whenever two memory references are to be disambiguated, an alias problem is formulated and solved. Propagation of AA information from Gimple to RTL has been addressed in [12] and implemented by a GCC patch and by a separate GCC branch. We have tried both implementations but were not able to obtain expected execution performance; in fact we observed a small performance decrease. This could have been caused by the following reason. In order to be effective for the case of partial loop unrolling on the RTL level, next to propagating the alias information from a GIMPLE representation (where the loop has not yet been unrolled), the algorithm would have to additionally disambiguate each newly introduced RTL memory statement. Such functionality was missing in the patch.

An alternative to propagating the alias information from Gimple to RTL is to enhance the AA on the RTL level. To achieve this we added flow-sensitive address-based alias analysis at the RTL level. Prior to our work, a similar approach has been proposed in [13]. The corresponding patch, however, has never been added to the GCC mainline due to associated increase in compilation time. Furthermore, this approach has the following drawbacks:

1. The technique is based on the idea of representing a memory address by means of an address descriptor, which is a pair \(<I,Z>\), where I is an operation and Z is a mod-k residue set. An address descriptor can keep track of only one operation (i.e., I). Due to this limitation, this approach is not able to disambiguate addresses obtained by linear combinations of values generated by more than one operation.

2. The technique extracts alias information across loop iterations, which leads to a significant increase in the compilation time. However, as pointed out in [11], the GCC internal scheduler deals with acyclic graph regions and, therefore, the extraction of alias information across loop iteration is of no use.

Our flow-sensitive address-based alias analysis overcomes the aforementioned limitations allowing to disambiguated memory accesses present within the same basic block. The analysis can be sketched as follows: Suppose a memory access part of an operation \(op\) which belongs to a basic block \(BB_s\). The address of the memory access, is given by an original linear function \(f\). By starting from \(op\) and traversing in reverse order the \(BB_s\) operations, \(f\) is composed with the linear expressions representing dependent operations. In the end we obtain a final linear function which represents the address in terms of regs defined outside the BB. Afterwards, the composition is continued over a number of control
paths; for each such path \( i \) a corresponding linear function \( f_i \) being derived. Those control paths are obtained as follows:

1. **non-backedge paths:**

   Starting from \( BB_i \) compose in reverse order over single predecessor blocks. Additionally we impose the limitation that among those single predecessor blocks at most one of them sources/sinks backedges. Once a BB with more than one predecessor is encountered, duplicate \( f \), one for each of the predecessors and continue for as long as single predecessor basic blocks are encountered none of those blocks sourcing/sinking back-edges.

2. **backedge paths:**

   Starting from \( BB_i \) compose in reverse order over single predecessor blocks. Once a BB with more than one predecessor over incoming backedges is encountered duplicate \( f \), one for each of the predecessors and continue for as long as single predecessor without incoming/outgoing back-edges blocks are encountered. Once a BB with more than one predecessor is encountered duplicate \( f \), one for each of the predecessors and continue for as long as single predecessor basic blocks are encountered, none of them sourcing/sinking back-edges nor the original \( BB_i \).

Once the linear functions \( f_i \) are derived for every memory access \( op \), two different accesses \( op_m \) and \( op_n \) from the same BB can be disambiguated. The two operations do not alias if: \( \forall i, f_i(op_m) - f(op_n) \neq 0 \).

**Example** Consider in Figure 1 a memory operation that belongs to the basic block \( E \). As a result of our CFG traversal 5 linear functions corresponding to 2 non-backedge and 3 backedge paths will be generated. Those paths are as follows:

the **non-backedge paths** consist of the following basic blocks: \{\( E, D, C, A, A1 \)\} and \{\( E, D, C, B, B1 \)\}.

the **backedge paths** consist of the following basic blocks: \{\( E, D, I, F \)\}, \{\( E, D, I, G \)\} and \{\( E, D, H \)\}.

Figure 1: CFG traversal during alias analysis: with red/blue you can see the non-backedge/backedge paths.

5 **Experimental Results**

The GCC ports for EVP and TriMedia have been compared to the existing compilers supporting these processors. For TriMedia, our tmGCC port has been compared to the production tmcc compiler which is a part of the TriMedia Compilation System (TCS) [6]. In the TCS toolchain, the core compiler tmcc generates the sequential code, and splits it into scheduling units, called decision trees (dtrees). The output of tmcc is then passed to a standalone VLIW scheduler tmsched [14]. According to the TCS convention, tmcc performs register allocation only for global registers which constitute a half of the complete register file. The tmsched scheduler performs register allocation of the remaining 64 local registers, peephole optimizations, and VLIW scheduling. Scheduled code then goes through the standard assembling and linking procedure to obtain the binary, which is then simulated or run on the target hardware to obtain the performance data. In order to generate scheduled TriMedia code with GCC, we follow a similar approach. Our tmGCC port acts as a core compiler generating sequential assembly, which is then formatted according to tmsched requirements, and passed to it for schedul-
ing. After scheduling, the final binary is generated in the same way as for tmcc.

We have experimented with a set of representative benchmarks from the media/signal processing domain. Our mediastone testsuite contains audiocodecs (ac3, mp3,dts), video algorithms (MPEG2, motion compensation), image processing kernels (color conversions and filters), and the complete EEMBC telecom suite. The perf-baseline suite contains several proprietary algorithms for image and video improvement, such as up-conversion, and custom implementation of MPEG-2 and h264 video codecs. We remark that these are product quality applications, where a significant part of the code was hand optimized by programmers.

In an embedded system, the most important factors for compiler evaluation are the performance and code size of the generated code, while the compilation time is not critical. Table 1 presents these metrics for the code produced using both GCC and the production compiler, tmcc. For brevity, we report the results only for the tm3271 core. The results for the tm3260, tm5250 and tm3282 cores were similar. In the table, tmGCC-old refers to the prototype TriMedia port developed at Philips Research earlier this decade. This prototype was not mature enough, lacking the ILP-enhancement features described in Section 4. It was relying completely on tmsched for ILP extraction and, essentially, can be seen as a port to a single-issue processor with TriMedia ISA. As depicted in the table, compared to tmcc, tmGCC-old exhibits severe performance degradations, 23.7% and 39.7%, showing that a bare GCC port without specific VLIW support is not well-suited for TriMedia, even when it is coupled to a mature VLIW scheduler. We remark that in addition to ILP-enhancement techniques, tmGCC-old is missing support for some of the addressing modes and for the new custom vector operations. This explains higher grade of degradation on perf-baseline, as a large portion of its applications was manually rewritten to utilize the new operations.

The numbers in the third column of the table represent the results obtained with tmGCC-current, the current port of GCC for TriMedia. We remark that in addition to the presented techniques, tmGCC-current contains support for vector operations missing in tmGCC-old and some enhancements to the GCC software pipelining and if-conversion passes. These techniques are currently under development by the TCS compiler team and are not reported in this paper. The presented results show that implementing the ILP-enhancing features exposes more ILP to tmsched, allowing it to dramatically improve the performance of the scheduled code. The performance gap with the mature production compiler is reduced to just 2.8% on mediastone and to 5.9% on perf-baseline. However, the performance improvement for mediastone is achieved at a cost of a 10% code size increase. We suppose that the loop-unrolling in our applications could have been too aggressive. The unroll factors in the benchmarks were chosen by the programmers to provide best performance with tmcc. These factors could be not optimal when compiling with tmGCC. Carefully selecting these factors would, probably, resolve this issue, but such work fell out of the scope of our study project. We have also identified another source of potential improvement for the GCC port. Over the years of development, a large number of peephole optimizations has been introduced to tmcc. Our GCC port lacks the vast majority of these peepholes. We remark that the GCC facilities for peephole optimizations have limitations which do not allow all the tmcc peepholes to be easily introduced. Namely, the peephole2 pass of GCC handles only adjacent operations. The operations which are non-adjacent, but connected by a data dependency can be handled by the combine pass. This pass, however, considers for optimizations only triples of operations connected by data dependencies, such that the dependence graph is linear. Allowing more generic forms of graphs would be desirable and could increase the power of peephole optimization pass of GCC.

<table>
<thead>
<tr>
<th>Testsuite</th>
<th>tmGCC-old</th>
<th>tmGCC-current</th>
<th>tmcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycles</td>
<td>123.7%</td>
<td>102.8%</td>
<td>100.0%</td>
</tr>
<tr>
<td>code size</td>
<td>100.7%</td>
<td>110.2%</td>
<td>100.0%</td>
</tr>
<tr>
<td>perf-baseline</td>
<td>cycles</td>
<td>139.7%</td>
<td>105.9%</td>
</tr>
<tr>
<td></td>
<td>code size</td>
<td>99.9%</td>
<td>102.5%</td>
</tr>
</tbody>
</table>

Table 1: Relative Performance of the GCC ports and the production compiler for TriMedia.

Similarly to experiments reported above, we have compared our GCC port for EVP with the current EVP production compiler. We remark that, differently from TriMedia case, our EVP port performs also VLIW scheduling. To achieve correct and efficient VLIW code generation, we have implemented the techniques presented in Section 3. The comparisons were performed on the standard benchmark for telecommunication industry,
EEMBC-telecom [8] and encouraging results have been obtained. Concerning our EVP GCC port (evpGCC), we would like to report an interesting experiment related to a comparative performance of compiler-optimized and hand-optimized code. In embedded systems the quality of the application code directly affects the cost and performance of the final product, thereby motivating significant amount of optimization effort done by programmers. This is particularly true for the telecommunication domain where programming using assembly or compiler intrinsics is still common. Although such programming model is costly, the numbers presented below provide a reason for this approach and motivation for improving the power of compiler technology. For our experiment we consider a 256-point complex FFT algorithm. First, a standard C implementation from EEMBC-telecom is taken, and is compiled using evpGCC. Second, we compile the tailor-made version of the algorithm version manually optimized for EVP [7]. Both implementations are simulated using the EVP simulator. The obtained results are depicted in Table 2, which shows that manual optimization provide a performance improvement by a factor of 122×.

<table>
<thead>
<tr>
<th>application</th>
<th>standard FFT</th>
<th>optimized FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>121086</td>
<td>986</td>
</tr>
<tr>
<td>cycles</td>
<td>3512</td>
<td>4936</td>
</tr>
</tbody>
</table>

Table 2: Performance and Code size for 256-point complex FFT.

A factor of 16 out of 122 can be explained by the inability of GCC’s vectorizer to vectorize the FFT code. In particular, the vector shuffle patterns employed in optimized implementation are hard to be auto-generated by the compiler. In fact, a factor significantly larger than 16 can be attributed to the absence of vectorization for the following reason. In the optimized version, the FFT butterfly data rearrangements are performed on the vector registers, whereas for the non-optimized code they are done via memory. This requires excessive memory traffic making the load/store unit a bottleneck. The dramatic performance gap suggests that, apart of vectorization, a number of other compiler techniques were not effective. We were able to identify several such cases. First, we observed that on the standard code, loop unrolling has not been performed by GCC, while in the optimized version the already vectorized loop body was further unrolled manually 4 times. The presented example, in our opinion, provides a challenging testcase for compiler engineers and may allow them to identify the weaknesses of existing optimizations.

6 Conclusions

The goal of the GCC porting projects presented in this paper was to evaluate suitability of GCC for code generation for non-interlocked VLIW processors. Our conclusions can be summarized as follows.

First, the obtained results illustrate that GCC can be used in a VLIW compilation toolchain, both as a core compiler coupled to an external VLIW scheduler, and as the complete solution performing both sequential code generation and VLIW scheduling. For TriMedia, which represents the former case, where our GCC port could benefit from a mature VLIW scheduler, the results were particularly encouraging, and the decision to productize our tmGCC prototype has been taken. Second, we have identified several areas where current GCC can be strengthened to better support VLIW compilation. Namely, GCC loop unrolling and alias analysis on the RTL can be improved to increase the amount of exposed ILP. Furthermore, DFA mechanisms in GCC have limitations when handling processor with significant number of instructions with several scheduling alternatives. Finally, GCC facilities for peephole optimizations, peephole2_optimize and combine have limitations, alleviating which could improve performance of both VLIW and non-VLIW targets.

In our ports we have developed partial solutions to some of the identified issues. However, development and integration of general solutions in the GCC framework will be of interest for the compiler engineers in the embedded domain considering to use GCC as a compiler framework for their VLIW (or non-interlocked pipelined) targets. In particular, our approach for inter basic block scheduling and for resource-aware delay slot scheduling can be improved.

In our opinion, some of the solutions implemented in the EVP and TriMedia ports could be beneficial to a wider range of GCC targets. First, the techniques which increase the amount of exposed ILP, such as the controlled loop unrolling and the addresses-based alias analysis on the RTL level, could be beneficial for non-VLIW processors that exploit ILP, e.g., superscalar or deeply pipelined scalar processors. Second, the DFA-related techniques presented in Section 3.4 allow dramatic reduction in the DFA size and generation time for the
cases when significant number of target instructions have multiple scheduling options and explosive growth of DFA is observed. Furthermore, in case the original DFA automaton for a processor has been factorized, our approach allows correct scheduling of instructions which contain alternatives from two different DFA automata.

Acknowledgments. We would like to thank our former colleagues from TriMedia TCS Compiler Team at NXP Semiconductors for the fruitful collaboration during the tmGCC study project and for generously providing us the experimental results for the current tmGCC port. We thank Jan Hoogerbrugge, Norbert Esser and Willem Mallon for their work on the initial tmGCC port. We also thank Wim Kloosterhuis for the discussions on EVP and for providing the production-quality FFT implementation optimized for EVP. Finally, we would like to thank Vladimir Makarov and other people in the GCC community for sharing their knowledge with us on the GCC mailing list.

References


Abstract

As the single-thread performance of general purpose processors no longer increases as quickly as it did in the past, system designers more and more look into alternative solutions to provide additional processing power to solve application problems. One trend that has been emerging, and proves to continue with future processors and systems that are currently under development, is to augment general processors with "accelerators" to take on special-purpose tasks. Depending on the system, these accelerators can be provided on the main processor chip itself, as an I/O-attached device, or as a separate system accessed via the network.

In some such hybrid systems, the accelerator elements are directly programmable by the user, resulting in applications that actually use more than one instruction set architecture at a time. Examples include the Cell Broadband Engine architecture with its Synergistic Processor Elements, and the use of graphics processors for general-purpose computing e.g. via the new OpenCL standard. To keep the complexity of software development for hybrid systems manageable, one important question is how such hybrid applications can be efficiently debugged. This paper examines how GDB can be employed to help address this challenge.

We will focus on the question how to adapt GDB to be able to debug applications running on more than one instruction set architecture. This requires fundamental changes to GDB internals that have been worked on for many years, and are close to completion at this point. Based on this effort, we will show how special solutions that were developed to address debugging the Cell Broadband Engine and hybrid systems based on it, like the Los Alamos "Roadrunner" supercomputer, can now be integrated back into mainline GDB. We will also explore how these capabilities can be extended to cover a broader range of hybrid multi-architecture debugging challenges in the future.

1 Hybrid Programming Models

Debugger support for hybrid systems heavily depends on the programming models used to implement applications for those systems. Thus we will begin by looking at some of the programming models used with hybrid systems currently on the market. Subsequently, we will investigate what features GDB would need to provide support for those models.

1.1 Cell Broadband Engine

The Cell Broadband Engine processor [1], used in the Sony Playstation 3 as well as IBM’s BladeCenter QS21 and QS22 systems, is a multi-core microprocessor, containing a Power Processor Element (PPE) and an array of eight Synergistic Processor Elements (SPEs). The PPE is a general-purpose 64-bit PowerPC-compatible processor providing the VMX multimedia extension. It runs the operating system and performs system management and control tasks; it can also run existing PowerPC-compatible application programs. Each SPE [2] consists of a Synergistic Processor Unit (SPU) and a memory-flow controller (MFC). They are intended to provide computational performance, in particular for game, media, and broadband workloads. The SPE implements a new instruction set architecture featuring a register file of 128 SIMD vector registers, each 128 bits wide. Load and store instructions allow to access a 256KB local-storage memory (local store) private to the SPE. All memory accessed directly by SPE load and store instructions, as well as the instruction text itself, must reside within that local store. However, the SPE is able to use DMA operations to copy data between local store and main storage, using facilities provided by the MFC.

Support for the Cell/B.E. processor has been available in the Linux kernel since version 2.6.16 as a subarchitecture of ppc64. In conjunction with the SPE Runtime Management Library libspe2 [4, 5], the spufs
virtual file system provided by the kernel allows applications running on the PPE to manage SPE contexts as virtual representations of SPE cores, load SPE applications into contexts and control their execution. A typical Cell/B.E. application will consist of a main component running on the PPE, and one or more compute components running on SPEs. The PPE part will assign tasks and distribute data to the SPE parts and coordinate their execution.

When building a Cell/B.E. application, the PPE and SPE components need to be compiled separately, using different tool chains [3]. The resulting SPE executables can either be loaded at run time, or else embedded at build time into PPE objects. The following code snippet shows the central steps required to execute SPE code:

```c
/* Reference embedded SPE program image. */
extern spe_program_handle_t program_spu;

/* SPE execution starts at ELF entry point. */
unsigned int entry = SPE_DEFAULT_ENTRY;

/* Create an SPE context. */
spe_context_ptr_t ctx = spe_context_create (0, NULL);

/* Load SPE program. */
spe_program_load (ctx,
    &program_spu);

/* Start executing SPE context. */
spe_context_run (ctx, &entry, 0,
    NULL, NULL, NULL);
```

Note that `spe_context_run` is a synchronous call that will return only after SPE execution has stopped; at this point, the `entry` variable will contain the local store address of the instruction that would have been executed next. This allows PPE code to implement routines that assist SPE execution; several such routines are already implemented in `libspe2` and will be transparently handled there.

In order to debug an application like that, the debugger needs to follow the flow of control between PPE and SPE code. The IBM SDK for Multicore Acceleration [6, 7] provides a version of GDB that supports this. However, this feature is not yet available in main-line GDB – in the rest of this paper we will examine what changes are required to contribute this capability upstream.

### 1.2 Roadrunner

The Roadrunner supercomputer at the Los Alamos National Laboratory was the first general-purpose machine to reach one quadrillion floating point operations per second (one petaflops) using the standard Linpack benchmark, and at the time of this writing still occupies the #1 slot on the Top500 list. The machine is a hybrid cluster based on IBM BladeCenter QS22 (Cell/B.E.) and LS21 (Opteron) blades. The basic hardware unit of the cluster is a compute node ("Triblade") consisting of one LS21 and two QS22 blades, with the two Cell/B.E. blades connected via PCI Express as "devices" to the Opteron blade. 180 such Triblades and 12 I/O nodes are clustered via Infiniband to form a "connected unit", and the whole Roadrunner system is made up of 18 connected units communicating via second-stage Infiniband switches. In total, the system thus comprises 6912 dual-core Opteron and 12960 Cell/B.E. processors.

Software running on Roadrunner needs to accommodate the memory and connection hierarchy of the system. A typical Roadrunner application will use MPI to communicate between the Opteron nodes of the cluster. On each LS21, a hybrid application uses the Data Communication and Synchronization ("DaCS") library [8] to start and manage tasks on the QS22 blades; each of those tasks will itself be a Cell/B.E. application using `libspe2` to manage computation on the blade’s SPE cores as discussed in the previous section.

DaCS maintains a hierarchical topology of processing elements ("DaCS elements"), which can serve the following roles:

- A general purpose processing element, acting as a supervisor, control or master processor. This type of element usually runs a full operating system and manages jobs running on other DEs. This is referred to as a Host Element (HE).
- A general or special purpose processing element running tasks assigned by an HE. This is referred to as an Accelerator Element (AE).
DaCS provides resource and process management services that allow an HE to manage AE resources below itself in the hierarchy, and initiate and manage execution of accelerated application on those AEs. In addition, DaCS provides communication services to manage, share, and synchronize data through remote DMA, message passing, and mailboxes between a HE and its associated AEs.

In the hybrid DaCS implementation used on a Roadrunner Triblade, the LS21 takes the role of the HE, while either the full QS22 blades or the Cell/B.E. processors on them are used as AEs. Both HE and AE run the Linux operating system, and a set of daemons is used to implement the DaCS process management services. Data transport is performed via the PCI Express bus.

To start an accelerated application on an AE, the HE application will use code along the following lines:

```c
/* Init DaCS library. */
dacs_init (DACS_INIT_FLAGS_NONE);

/* Reserve a DE for use. */
uint32_t num = 1; de_id_t de;
dacs_reserve_children (DACS_DE_CBE, &num, &de);

/* Start application on DE. */
dacs_process_id_t pid;
dacs_de_start (de, "program_cbe", NULL, NULL,
   DACS_PROC_LOCAL_FILE, &pid);
```

Note that `dacs_de_start` identifies the application to be executed via a file name on either the local or the remote system, or as an embedded SPE executable (only supported when the HE is a Cell/B.E. system). The routine is asynchronous and returns a handle that can be used to manage further execution of the program on the AE.

As in the case of a Cell/B.E. application, a hybrid application using DaCS contains tightly integrated components that interact via shared memory, mailboxes, and other communication and synchronization primitives. Therefore, an integrated view of all those interconnected components during a debug session is very helpful. Unfortunately, GDB today does not support integrated debugging of a DaCS application – you need to use separate GDB sessions to debug the HE and AE parts. To make this a bit easier to use, we have developed an internal prototype application of a "DaCS GDB Manager" that automates the process of setting up those GDB sessions and presents a unified user interface based on a `screen` session encapsulating them.

At the start of a debugging session, the DaCS GDB manager will set up a single GDB on the HE and run the main application under it. By hooking into a debug service provided by the DaCS daemons, the DaCS GDB manager is aware whenever an accelerator application is started on an AE via `dacs_de_start`, and intercepts the call to insert a new GDB session debugging the new application (using the remote gdbserver interface to execute it on the AE). In addition, the DaCS GDB manager hooks into the remote protocol connection between each GDB session it manages and the corresponding remote gdbserver, so that it becomes aware whenever some event happens on the remote side (e.g. a breakpoint is hit). At this point, the `screen` session is requested to bring the corresponding window to the foreground.

However, an even better solution would be to integrate support for hybrid debugging into GDB itself. We will investigate how this goal could be achieved based on current mainline GDB. Beyond support for a single hybrid DaCS application, it would be interesting to see whether it is possible to support debugging a whole MPI cluster application in a single GDB session. Today this seems to be only supported by GUI tools like the Eclipse Parallel Tools Project (PTP).

### 1.3 OpenCL

Over the past several years, developers have started using the capabilities of graphics processors to perform not only graphics rendering itself, but to accelerate general-purpose processing. The programming models used in this approach have typically been vendor-specific. Recently, however, the OpenCL specification [9] is emerging as a common programming model that would allow code to be portable not only across different graphics processors, but multiple types of compute accelerators in general. For example, we are currently
investigating providing support to run OpenCL kernels on Cell/B.E. SPEs.

OpenCL is being created by the Khronos Group with the participation of many companies and institutions. Version 1.0 of the specification has been publicly released in December 2008.

The OpenCL platform model describes a host system with one or more compute devices. An OpenCL application runs on the host system, but schedules compute kernels on available compute devices for execution. The primary programming model is data-parallel, that is the same kernel executes in parallel over an index space called NDRRange (n-dimensional range, where \( n \) equals 1, 2, or 3). Optionally, OpenCL implementations may in addition support a task-parallel programming model.

OpenCL compute kernels are written in a new OpenCL programming language, which is based on C with a certain set of restrictions and extensions. These are typically compiled at run-time to support execution on whatever compute device the current host has available. The OpenCL runtime provides services to build and manage program and kernel objects, which can then be scheduled onto command queues to be executed in compute device contexts. A typical application will use code along these lines:

```c
/* Query compute devices. */
cl_device_type type
  = clGetDeviceType(CL_DEVICE_TYPE_ACCELERATOR);
cl_device_id dev; cl_uint num;
clGetDeviceIDs(type, 1, &dev, &num);

/* Create context. */
cl_context ctx
  = clCreateContext(NULL, num, &dev, NULL, NULL, &err);

/* Create command queue. */
cl_command_queue command_queue
  = clCreateCommandQueue(ctx, dev, 0, &err);

/* Build OpenCL program. */
const char *strings
  = "__kernel void kernel() {\n"
  /* program source code */\n  "}\n";
cl_program program
  = clCreateProgramWithSource
      (ctx, 1, &strings, NULL, &err);
/* Compile OpenCL program. */
clBuildProgram(program, num, &dev,
               NULL, NULL, NULL);
/* Extract compute kernel. */
cl_kernel kernel
  = clCreateKernel(program,
                   "kernel", &err);
/* Execute kernel on NDRRange. */
size_t global_size[1] = { 5 };
size_t local_size[1] = { 1 };
cl_event event;
clEnqueueNDRangeKernel
    (command_queue, kernel, 1, NULL,
     global_size, local_size,
     0, NULL, &event);
```

As compute kernel routines get more and more complex, it becomes important to be able to debug this code as well. Of course, it depends on the hardware capabilities of each specific compute device to which extent debugging is even possible. Note that even if debugging the intended target device does not work, OpenCL implementations should allow defining the main host processor as "compute device" for debugging purposes. If the compute device allows debugging (as e.g. Cell/B.E. SPEs would), exploiting this support in GDB poses similar challenges to the models described above.

## 2 GDB Internals Overview

A common property of the hybrid programming models we have described in the previous section is that in addition to the main application, one or more accelerator applications or compute kernels are executing. Depending on the model, execution may be:

- synchronous to a main application thread;
- modeled as an independent thread of control within the main application;
modeled as an independent process running on the same system as the main application; or

- running as a process on a different system.

In either case, compute kernels will generally (but not always) be implemented in machine code for a different processor architecture. The executable format used to store this code and associated debugging information (if any) may also be different. Nevertheless, from the programmer’s point of view compute kernels are closely integrated with the execution of the main application, and interact using shared memory and other communication and synchronization mechanisms. This makes it an interesting goal to support integrated debugging of a complete hybrid application within a single GDB session.

In order to understand how GDB needs to change in order to achieve this goal, let us start by looking at the internal structure of today’s GDB [10]. There are three major subsystems: user interface, symbol handling, and target system handling.

The user interface consists of three distinct interfaces GDB provides: the command-line mode, a curses-based windowed text interface, and a machine interface intended to enable GDB to be used as the back-end engine for other debuggers (e.g. Eclipse).

The symbol side of GDB handles reading object files, interpreting debugging information, managing symbol and type information, and parsing, evaluating, and printing expressions in a variety of source languages. Most features of the symbol side are available even in absence of a live program (or core file) to debug, and require solely the presence of a executable file.

The target side of GDB handles control of an actual program: starting, attaching to, or killing a process to be debugged, execution control (stopping, continuing, or single-stepping), accessing target registers and memory, unwinding the call stack, and supporting breakpoints, watchpoints, and tracepoints. GDB supports many different methods of handling a target, including processes running on the same machine as GDB itself ("native" targets), remote machines attached via the network or serial lines (remote targets), and post-portem debugging on core files.

Both the target side, and to a lesser extent the symbol side, require information about the machine architecture and ABI details used by the program to be debugged. The target architecture (gdbarch) provides these details about:

- Data types and representation: size and other properties of standard C data types on the platform, byte order, floating-point format, C++ virtual function pointers.
- Machine addresses: encoding and decoding addresses into target pointer values, address spaces.
- Machine code disassembly and analysis: disassembling the instruction set architecture, skipping function prologues and glue code stubs.
- Registers: register numbers, names, and default types; data representation of register contents, special registers (PC, SP, etc.).
- Frame handling: stack unwinding, handling of special cases like signal handlers.
- Function calling convention: encoding function arguments for inferior calls, decoding return values.
- Run control details: installing and handling breakpoints, implementing single stepping.
- Shared library handling: detecting and handling shared libraries managed by the target process.
- Core file handling: detecting core file architecture, retrieving register contents and shared library data.

A central problem in supporting hybrid multi-architecture debugging is that many of the above properties used to be hard-coded into GDB, either in the form of configuration settings selected at compile-time, or in the form of global variables retaining state accessed throughout GDB code that is implicitly assumed to never or rarely change. For example, GDB used to (and partially still does) hold global state describing:

- the target architecture (current_gdbarch)
- the target vector (current_target)
- the inferior process (inferior_ptid)
- the source language (current_language)
- the main executable file (exec_bfd)
While work has been ongoing to remove this global state and transform GDB internals so that they could more naturally represent hybrid application scenarios, this has been a slow and complex process. In the following sections, we will review what has already been achieved and what still needs to be done.

3 Multi-Architecture Support in GDB

3.1 From Target Macros to gdbarch

The process of allowing simultaneous support for multiple target architectures in GDB has been ongoing for over a decade at this point. Early versions of GDB, up to the GDB 4.x series of releases, required all target architecture information to be selected at compile time. Depending on the --target= configuration option, a Makefile fragment config/target/target.mt was added by the configure step to the main GDB Makefile. This fragment would set a variable MT_FILE to identify a header file, typically with a name along the lines of config/target/tm-target.h, which was then automatically included during the build of most GDB files. In this target header file, properties of the target architecture were hard-coded by providing macro definitions like:

```c
#define TARGET_LONG_BIT 64
```

If values like this are selected at compile-time, it is clear that no single build of GDB would be able to support both 32-bit and 64-bit variants of an architecture at the same time.

GDB 5.0 was the first step in the direction of multi-architecture support. It introduced the concept of a gdbarch structure, which is a data structure collecting the information that used to be provided via the above target macros. A target-specific source file would provide routines to allocate a gdbarch and initialize it with values appropriate for the architecture. This made it possible to support multiple such initialization routines in a single GDB, which would use properties of the target and/or the executable file to determine which gdbarch to use — however, at this time, an single GDB build was able to support only different variants of the same basic processor architecture (e.g. 32-bit and 64-bit variants, or different OS versions on the same architecture).

As an example, instead of defining the target macro TARGET_LONG_BIT, the MIPS target would provide gdbarch variants to support either 32-bit or 64-bit target processes:

```c
static struct gdbarch *
mips_gdbarch_init (info, arches)
    struct gdbarch_info info;
    struct gdbarch_list *arches;
{
    struct gdbarch *arch;
    [...]arch = gdbarch_alloc (&info, [...]);
    [...]switch ((elf_flags & EF_MIPS_ABI))
        case E_MIPS_ABI_EABI32:
        set_gdbarch_long_bit (arch, 32);
        [...]break;
        case E_MIPS_ABI_EABI64:
        set_gdbarch_long_bit (arch, 64);
        [...]break;
    }
    [...]return arch;
}
```

To avoid having to convert all architectures supported by GDB at the same time, the transition was gradual: Common code would continue to use the old macros. However, a special glue layer would re-define all macros not defined by the current target header file to internally access a global current_gdbarch structure:

```c
extern int gdbarch_long_bit
    (struct gdbarch *gdbarch);
extern void set_gdbarch_long_bit
    (struct gdbarch *gdbarch,
     int long_bit);
#if GDB_MULTI_ARCH
#if (GDB_MULTI_ARCH > 1) \n    || !defined (TARGET_LONG_BIT)
#define TARGET_LONG_BIT \n    (gdbarch_long_bit
     (current_gdbarch))
#endif
#endif
Note that a special complication needed to be addressed: GDB common code would refer to those macros in initialization code that executes only once. For example, _TARGET_LONG_BIT was used in GDB startup code to define a global variable builtin_type_long which represents the target’s long data type. This no longer works if current_gdbarch is reset after a different executable file is loaded into GDB. However, these global variables were themselves used throughout GDB common code and were difficult to eliminate. Therefore, as another intermediate step, a mechanism was established to recompute the values of such variables whenever current_gdbarch changes. Of course, this constitutes another obstacle to use multiple architectures at the same time.

Over the course of the GDB 5.x release cycles, all targets supported by GDB were converted to use the new gdbarch logic (or, in some cases, support for old targets was removed). GDB 6.0 was the first release where all targets used the new architecture framework. During the following GDB 6.x release cycles, further updates were made to the register and frame handling infrastructure, and a number of backwards-compatibility hacks to support old code were removed.

### 3.2 Support for --enable-targets

At this point, every target had been converted to use the new architecture framework, but it still was not possible to build a single GDB that supported more than one different processor architecture. There were yet a number of obstacles to overcome: Some architectures were still defining target macros in a target header file, because a single setting was appropriate for all configurations of the processor architecture, or in some cases, because some rarely used macro had not even been converted yet into a gdbarch setting. In addition, some architectures were using the target header files and/or makefile fragments for special purposes like including extra header files or adding compiler flags.

Over time, we managed to clean up those remaining issues until every single target header file was completely empty and could thus be removed, and the only remaining purpose of the target Makefile fragments was to list object files making up support for this particular target architecture:

```bash
TDEPFILES= \
```

The only remaining piece to support multiple target architectures was to enable building GDB while including more than one such set of target files into the build. Amongst other changes, this required a rework of the shared library support files to enable building support code for more than one type of shared library into a single GDB — today, the gdbarch selects which of the built-in shared library handlers is to be used with the given target. The remaining information was moved from Makefile fragments into a single configure. tgt file, which can be parsed multiple times to accumulate information for different target architectures.

As of GDB 6.8, the current GDB release, this process has been completed. GDB now supports a configure option --enable-targets that can be used to specify additional targets that should be supported besides the main target configured via --target. For example, a debugger configured with

```bash
./configure \
--host=powerpc64-linux \n--target=powerpc64-linux \n--enable-targets=arm-elf,mips-elf
```

will support debugging native application on PowerPC, but at the same time support connecting to and debugging applications running on remote ARM or MIPS machines (or core files from such applications). As a special case, --enable-targets=all will include support for all target architectures supported by GDB into a single binary.

### 3.3 Removal of current_gdbarch

While it is now possible to build support code for multiple architectures into a single GDB binary, this is still not sufficient to implement hybrid debugging scenarios like required e.g. for Cell B.E. — the reason for this is that we still have the global current_gdbarch variable. Just about every part of GDB accesses this variable to retrieve architecture-specific data; while the global can be reset at defined points in time, e.g. whenever a new connection to a target is opened, this approach cannot handle hybrid situations where two or
more architectures need to be handled simultaneously. The only way to achieve this is to completely remove current_gdbarch.

We have already made significant progress towards that goal. In particular, GDB no longer uses "swapped data" that must be reloaded whenever current_gdbarch is changed — note that eliminating the two major users of this feature, current_regcache which was used throughout GDB to access the "current" register contents, and builtin types like the builtin_type_long global mentioned above, in itself required significant rework. In addition, as there are no more target macros, all places where target macros were used within GDB have been rewritten to directly access the appropriate gdbarch member, so that there are no more "hidden" uses of current_gdbarch through macro definitions in header files.

Still, this leaves us with a significant number of explicit references to the global that need to be removed — and replaced by whatever architecture is the appropriate one in a hybrid setting. In some places, this is obvious, e.g. code implementing gdbarch callback routines will always refer to the architecture for which it was invoked. However, in many places code using current_gdbarch needs to be modified to make an explicit choice. Often, this involves making some GDB data structure architecture-aware by adding a gdbarch pointer as member.

For example, GDB's regcache and frame_info data structures holding information about a target's current register contents and stack frames are clearly related to the architecture defining the target's register set. Similarly, a GDB expression that was evaluated in a specific context (i.e. while the user was examining a particular stack frame) needs to retain the related architecture information to enable evaluation of that expression using correct per-architecture settings. The same applies to constructs like breakpoints and watchpoints.

Work on these changes is still in progress. At the time of this writing I have a set of patches that allow building a debugger that no longer uses a current_gdbarch global variable. However, there is still some more work required to get these into a shape acceptable for mainline integration. At this point, I'm hoping to be able to conclude this work in time for the next major GDB release.

3.4 Cell/B.E. multi-architecture support

Once current_gdbarch is gone, we will be able to contribute GDB support for full Cell/B.E. debugging to mainline. (The current out-of-tree implementation available in the SDK tries to work around the current_gdbarch problem by resetting that variable at selected points. While this works most of the time, it is not a solution that would be acceptable for GDB mainline.)

The central infrastructure piece required for Cell/B.E. debugging is per-thread and per-frame architecture selection. As SPE execution within a Cell/B.E. process is synchronous and happens while the PPE thread of control is blocked within a spu_run system call, we need to detect whether this is currently the case. Using a new target vector method target_thread_architecture GDB common code will be able to determine the architecture to be used for a given thread. The Cell/B.E. implementation of this call will inspect register values and the current instruction to determine whether the thread is in fact executing spu_run and use the information to return either SPE or PPE as current architecture. As GDB already tags its register caches with their associated architecture, platform-specific code for Cell/B.E. will then be able to operate on either a PPE or an SPE register set for the given thread. (The PPE register set is available always, while the SPE register set is available in addition during SPE execution.)

However, just because a thread is currently executing SPE code does not imply that its whole execution history is on the SPE. In fact, we want the Cell/B.E. debugger to extend stack backtraces beyond the top of the SPE stack to show the PPE code that triggered SPE execution by issuing the spu_run system call. On the other hand, a thread might currently execute PPE code which implements an assisted call triggered from the SPE; we want the backtrace to model this relationship as well. To implement this feature, GDB common code will extend its stack frame unwind logic by tagging each stack frame with its associated architecture, and allowing platform-specific unwinder code to report that the previous frame in the stack chain has a different architecture than the current one. Both SPE and PPE platform code would provide such architecture unwinders to properly model the transitions described above. As an example, the following shows the backtrace of a thread that is executing...
a PPE-assisted call invoked from SPE code that was in turn started from other PPE code:

(gdb) backtrace
#0 0x0fedec28 in __nanosleep_nocancel ()
    from /lib/libc.so.6
#1 0x0fe0c028 in ?? ()
    from /usr/lib/libspe2.so.2
#2 0x0fe0ab80 in spe_default_posix1_handler ()
    from /usr/lib/libspe2.so.2
#3 0x0fe0eca4 in spe_handle_library_callback ()
    from /usr/lib/libspe2.so.2
#4 <cross-architecture call>
#5 0x0003fee4 in ?? ()
#6 0x000006c0 in nanosleep ([...])
    at libgloss/spu/nanosleep.c:51
#7 0x0000026c in sleep ([...])
    at libc/machine/spu/sleep.c:15
#8 0x0000001a4 in main ([...])
    at program-spu.c:8
#9 0x0000008c in _start ()
    from program-spu@0x10001b00 <5>
#10 <cross-architecture call>
#11 0x0fe10238
    in _base_spe_context_run ()
    from /usr/lib/libspe2.so.2
#12 0x0fe051c0
    in spe_context_run ()
    from /usr/lib/libspe2.so.2
#13 0x0100015f0 in main ()
    at program.c:21

The final piece that needs to be added to GDB to support Cell/B.E. debugging is support to find and use the executable images associated with SPE code, in addition to the main executable and shared libraries running on the PPE side. This is implemented by registering the SPE images as additional "shared libraries" using a new shared library handler that PowerPC target code will install whenever it detects the target processor is in fact a Cell/B.E. GDB will directly access the copies of the SPE ELF images present in the target program’s PPE address space; this allows transparently supporting embedded SPE images as well as external files opened at run time via spe_program_load. As all this code is already available, I expect to be able to commit this feature to mainline shortly after current_gdbarch is eliminated.

4 Related Work

In addition to the multi-architecture support effort described above, there are several additional areas where GDB needs to be extended in order to support various other hybrid scenarios. In particular, to fully support debugging a "Roadrunner" application, it should be possible for GDB to simultaneously attach to more than one process, running on more than one machine.

There are several restrictions in current GDB that prevent this from being possible:

- Multiple inferior processes. GDB 6.8 supports only debugging a single process at a time. The next version of GDB will add support for debugging more than one process, but only under certain restricted circumstances (in particular, all processes must run the same executable).

- Multiple target connections. Current GDB always connects to a single target at one time. In the Roadrunner setup, we would like GDB to simultaneously debug a program running on natively on the current system, and two remote applications running on different remote machines. This will require eliminating the current_target global.

- Multiple executable files. GDB today supports just a single main executable exec_bfd. This should be extended to support different main executables for each inferior process; symbol lookup needs to take those into account.

- Multiple address spaces. GDB uses a single scalar value of type CORE_ADDR to uniquely describe an address in the target process. (The Cell/B.E. patches work around this problem by "encoding" PPE and SPE addresses into a single range of values to be used as CORE_ADDR.) When using multiple inferiors, this value by itself is no longer sufficient; code referring to a CORE_ADDR needs to always take the correct inferior into account.

An effort to address many of these issues is currently under way [11]. In particular, code to implement support for multiple executables and address spaces as well as
improved support for multiple processes is available in multiprocess-20081200-branch in the GDB CVS repository today. Support for multiple target connections is still an open issue. For further discussion of the multi-process work, see also Marc Khouzam’s article in these proceedings.

5 Conclusion

The decade-long effort to provide multi-architecture support is nearing completion. A new effort to provide multi-process support is under way and already shows significant results. Once these new features are available in GDB, we should be able to combine them to provide full support for debugging applications using new hybrid programming models including libspe2 on the Cell Broadband Engine, the hybrid DaCS model used on Roadrunner, as well as the new OpenCL model. As a first result, we expect that Cell/B.E. multi-architecture debugging capabilities will become available with the next major GDB release.

References


