Enabling Interactivity in GCC for Fine-Grain Optimizations

Cupertino Miranda*, Grigori Fursin*,
Sebastian Pop*§, Albert Cohen*[1]

* ALCHEMY Group, INRIA Futurs and LRI,
Paris-Sud University, France (HiPEAC members)
§ Advanced Micro Devices, Inc.

ABSTRACT

Current state-of-the-art compilers often fail to deliver best possible code on rapidly evolving hardware due to hardwired optimization heuristics and inability to fine-tune applications for best performance, parallelism, power consumption, code size and other constraints. Recently, we introduced an Interactive Compilation Interface (ICI) for PathScale compiler (Open64) and GCC to tackle the above problems by instrumenting the parts of the compiler where decisions about fine-grain transformations are made. Though this solution enables quick prototyping of the system to experiment with automatic program tuning using iterative compilation and machine learning, it has potential extendibility and portability problems when keeping up-to-date with parallel GCC developments. To solve these problems, we suggest to introduce an intermediate data layer to GCC (GDL) to separate program analysis and transformation phases, and pass/synchronize all necessary optimization information through GDL. This should simplify development and updates of compiler procedures responsible with external optimization drivers. We believe that this will also help to modularize internal GCC structure and move towards pluggable compiler architecture with automatically tunable compiler heuristics.

KEYWORDS: interactive compilers, interactive compilation interface, iterative compilation

1 Introduction

Fine-grain program optimizations are often used to improve program performance on top of current state-of-the-art compilers [BKK+98, CSS99, FOK02, KZM+03, TVA05]. Recently, we modified PathScale compiler (Open64) and GCC to enable interaction with the external optimization drivers to be able to modify internal hardwired optimization heuristics and

[1]E-mail: {cupertino.miranda,grigori.fursin,sebastian.pop,albert.cohen}@inria.fr

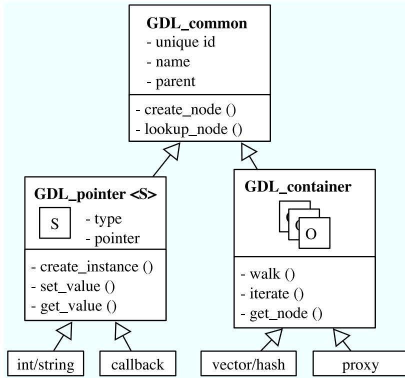

Figure 1: GDL interface

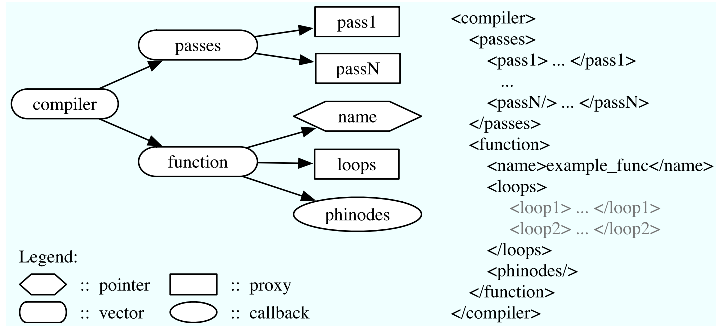

Figure 2: GDL use example

fine-tune programs for various constraints (performance, power, code size, parallelism, design space exploration, etc) [FC07]. By unifying and using Interactive Compilation Interface (ICI) we expect to build modular compilers with pluggable transformations that can learn optimization heuristics automatically and continuously on rapidly evolving architectures. Unfortunately, current implementation of the ICI is also hardwired with the internal compiler optimization which makes it difficult to port to newer versions of the compiler. To solve this problem, we introduced an intermediate data layer for GCC (GDL) that is used to uncouple application of transformations for the program analysis (decision making). All necessary information about optimizations is now passed through GDL to avoid hardwiring of ICI instrumentation into compiler optimization heuristic thus considerably simplifying ICI implementation, extendibility and interaction with the external tools.

2 GCC Data Layer Framework

The unique way to include a new transformation pass in GCC was by writing C code tightly sealed within GCC's framework, using the data structures and access functions of GCC's intermediate representations. This incurred difficulties to write reliable tools that could interact with GCC, such as "smart text editors" that could benefit of one of the best C++ parsers available on the market, stand alone static analyzers that could benefit of the advanced intermediate representations of GCC (CFG, SSA, call graph, etc.), or more generally stand alone pluggable passes. In order to address the stability problem of GCC's interfaces, we propose the GCC Data Layer to uncouple passes from the format of the data they are using. This data layer provides not only independence of the underlying data structures, but also an interface for interactively observe and modify the data processed by the compiler.

2.1 Interface of GDL

We first describe the components building GDL, the interface that one can use to access GDL, and then a concrete study of a test case. GDL is composed of three main objects: `GDL_pointer`, `GDL_container`, and an object that gathers common parts of these two objects, `GDL_common`. The relations between these objects is illustrated in Figure 1.

- `GDL_common` provides the common parts for `GDL_pointer` and `GDL_container`, `id` uniquely identifies each GDL object, and `parent` connects to other GDL tree node,

creating relationship. `lookup_node (path)` returns pointers to the object reachable by a string containing a `path` to an object. `path` is a composition of an `id` and `name`'s representing a single GDL node, like a path specified in the XPath language.

- `GDL_pointer` contains a `pointer` to memory allowing it to be accessible. It is then extended to contain `type` information.
- the `GDL_container` collects a set of objects and provides the interface to walk and iterate over these objects: `walk (node, walk_helper)` - walks through `node` and its sub-nodes and `walk_helper` contains a call-back function and its input data. `proxy` is an extension of `GDL_container` allowing maps to GCC structures.

To create more specific behavior in GDL framework, `GDL_pointer` functionality is extended to `integer` and `string` nodes, and `GDL_container` is extended to vector and proxy nodes. `integer` and `string` nodes are capable of storing references to simple types, `vector` node makes it possible to group several types of nodes in a single node, and `proxy` nodes map GCC data structures allowing GDL to read/write from/to these structures. GDL nodes are kept in a tree structure in which the leaves keep type information and pointers to data. This tree structure organizes data hierarchically simplifying implementation of generic walkers and enables exporting data to formats such as XML.

Proxy nodes contain only a pointer to the head of a data structure and no other sub-nodes. For example, when storing a graph only the entry node is stored. Whenever the sub-nodes are accessed (or traversed), the GDL framework dynamically allocates and initializes these sub-nodes. Special care must be taken with `proxy` type nodes since memory leakage can occur from bad usage and when the structure contains data recurrence, infinite loop can occur, i.e. in graphs. Memory leakage can be controlled if a garbage collector is used.

2.2 Usage examples

We now consider the construction of an external system for static analysis and transformation using the GDL framework. First, a part of the structures of GCC are exposed to the external tool using GDL data structures. Then a call to the interactive framework is inserted at strategic points (pass managers, between analysis and optimizations, etc.), such that the compiler relies on the external tool to take decisions. At these points, the compiler control flow is stopped, and the external tool can observe and modify the data structures.

Figure 2 shows a small example of the internal data inside GDL and the possible XML for it. GDL nodes content is being constantly updated during the compilation and for this reason, GDL should always be accessed atomically, while the compiler process is locked.

Interactivity frameworks generally also require the possibility to change data inside the compiler. For instance, ICI could be connected to a compilation driver which would decide, by code analysis, to customize the `unroll factor` of a loop. GDL provides a way to update data instead of only reading it. For this reason, every node contains an identification code (`id`) which is also added to a hash table to minimize lookup performance. In case of a `proxy` node, its sub-nodes cannot have an `id` since they are dynamically generated whenever the node is visited. To solve this issue, the GDL nodes also provide a path notion. For example, if the node `loops` in Figure 2 contains `id = 42` and the field `unroll_factor` which is inside the node `loop2` must be updated, the node `unroll_factor` can now be referenced with the path `42/loop2/unroll_factor`.

2.3 GDL and GCC Integration

In order to expose some GCC data structures using the GDL framework, the only thing to modify in the compiler is at the allocation and deallocation of that data structure by registering a pointer to that data in a `proxy` node. GDL has no impact in the compiler running time and memory consumption when GDL is not externally accessed. Accessing GDL locally in the compiler should be avoided, as there is an overhead to directly access the data: i.e. building and accessing the wrapping structures.

3 Future work

GDL can clean and simplify interactivity frameworks, as it standardizes the interface to access the data structures of the compiler. As a first concrete application, it is possible to implement an external pass reordering tool by using a `proxy` node for registering the root node of the passes structure that defines the static dependences between passes. Another application of GDL is to provide a practical interface to create interactivity with GCC via scripting languages.

One possible improvement in the usability of GDL is to use the garbage collector of the compiler to register root pointers, and data structure layouts.

References

[BKK+98] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, 1998.

[CSS99] K.D. Cooper, P.J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.

[FC07] Grigori Fursin and Albert Cohen. Building a practical iterative interactive compiler. In *1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), colocated with HiPEAC 2007 conference*, January 2007.

[FOK02] G.G. Fursin, M.F.P. O'Boyle, and P.M.W. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.

[KZM+03] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 12–23, 2003.

[TVA05] S. Triantafyllis, M. Vachharajani, and D. August. Compiler optimization-space exploration. In *Journal of Instruction-level Parallelism*, 2005.