WHOPR - Fast and Scalable Whole Program Optimizations in GCC

Initial Draft

12-Dec-2007

Preston Briggs Doug Evans Brian Grant Robert Hundt William Maddox
Diego Novillo Seongbae Park David Sehr Ian Taylor Ollie Wild

Google

Abstract

This document describes some initial design ideas on a whole program optimizer infrastructure for GCC. None of the ideas discussed here should be considered set in stone. This is only a draft to be used for discussion purposes. We expect many details to change over time, but the basic structure should be flexible enough to accommodate an efficient implementation.

Introduction

Whole-program analysis increases optimization opportunities. Since the compiler has visibility over every function in every compilation unit, decisions that would normally use conservative estimates can instead be based on data-flow information crossing file boundaries. Additionally, the compiler is able to perform cross-language optimizations that are not possible when the compilation scope is restricted to individual files.

On the other hand, a basic implementation of whole program optimization has the potential to incur massive memory consumption during compilation. Since every function body in every file may be needed in memory, only relatively small programs will be able to be compiled in whole program mode. This document describes WHOPR (**who**le **p**rogram optimize**r**), a whole-program optimization framework for GCC with the following goals:

1. It can handle arbitrarily large programs (in the order of millions of functions)
2. Optimization in whole program mode operates in time and memory complexity comparable with optimization of individual files.

3. WHOPR can accommodate several modes of operation to improve compile times and memory utilization
    1. Call-graph partitioning to group closely related functions
    2. Support incremental optimization

The design is based on the following premises:

1. It is impossible (or undesirable) to fit all the function bodies in memory.
2. The global call-graph itself can always fit in memory. This will ultimately limit scalability, but it should be possible to handle call-graphs with ~1M nodes and ~1M edges in less than 500Mb of memory.

Although the approach suggests that the compiler will have access to every single component that makes up the final application binary, in reality this situation will rarely, or never, occur. Modern software systems make extensive use of third party and system libraries that are dynamically loaded while the application executes. When these boundaries are found during optimization, GCC could simply handle the case as a regular clobber/escape site. Other, more aggressive, alternatives are possible as well and discussed in later sections.

Design

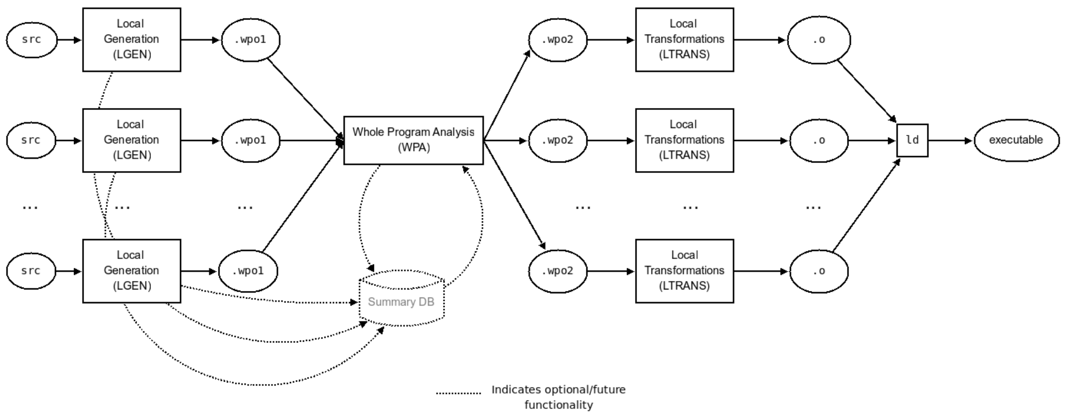
WHOPR tries to maximize the amount of parallel and independent work to take advantage of clustered and multiprocessor machines. Optimization proceeds in three main phases

1. Local generation (LGEN)
   This stage executes in parallel. Every file in the program is compiled into intermediate language (IL) together with the local call-graph and summary information.

2. Whole Program Analysis (WPA)
   This stage is performed sequentially. The global call-graph is assembled, and a global analysis process makes transformation decisions. The global call-graph may be partitioned to facilitate optimization during phase 3.

3. Local transformations (LTRANS)
   This stage executes in parallel. All the decisions made during phase 2 are implemented locally in each target file, and final object code is generated. Optimizations which cannot be decided efficiently during phase 2 may be performed on local call-graph partitions.

This process is depicted below. Notice that the files emitted by the LGEN phase are not necessarily the same files as those used during transformation. The details of each phase are described in the following sections.


src
Local Generation (LGEN)
.wpo1
Whole Program Analysis (WPA)
Summary DB
.wpo2
Local Transformations (LTRANS)
.o
ld
executable
...
Indicates optional/future functionality

Overview of WHOPR

Another important issue to consider is that not every aspect of this design needs to be implemented inside GCC. While each of the three main stages (LGEN, WPA and LTRANS) are to be implemented inside GCC, the launching of processes and the distribution of files will probably be implemented as an external helper application.

Local generation (LGEN)

During this stage, every compilation unit (CU) is compiled into an intermediate object file, containing the GIMPLE representation and summary information needed for the WPA phase. For notational convenience in this document, we will assume that from each input file a file with extension **`wpo1`** is generated (the implementation may decide to use a different extension).

The contents of **`wpo1`** files may vary and it is organized in two broad areas: an index section, with summary information for the file, and a body section, with the GIMPLE representation of every function in the file. The index section contains everything needed to analyze the file without having to traverse all the function bodies. Minimally, it should contain:

1. Local call-graph, or at least all the call-graph nodes for every function in the file.
2. Symbol table, describing every global and local symbols defined in the file.
3. Type table, describing every data type referenced in the file.

An important implementation feature in this phase will be streaming speed. The WPA phase will need to very quickly incorporate and aggregate local information from every compilation unit to build the global call-graph and perform analysis. To this end, an important implementation goal for **`wpo1`** files is compactness of representation to provide fast IR and index streaming.

Additionally, to support incremental analysis in the future, a global database may be maintained with summary information for all the files in the application. This database need not be large. One alternative is for each LGEN process to simply write checksum information for the body into the Incremental IPA database. This checksum can later be used by WPA to determine whether a particular file has changed since the last time the application was compiled.

Another alternative for LGEN is to compile each file to completion, and emitting locally optimized target code in each **`wpo1`** file. This would allow the compiler to bypass the WPA and LTRANS phases and go directly to the linker.

Whole Program Analysis (WPA)

After all the individual CUs have been compiled and (optionally) optimized locally, the compiler collects all the local call-graphs from every CU to perform whole program analysis (WPA) on the global call-graph. This will produce a global optimization plan that will later be used to perform local transformations (LTRANS).

The global call-graph is the union of all the individual call-graphs in every CU. Summary information in each local call-graph is propagated over the global call-graph and a transformation plan is decided. Conceptually, this phase writes out new object files (**`wpo2`**), containing a transformation plan and the analysis results from WPA that affect the local call-graph. However, the mapping between **`wpo1`** and **`wpo2`** files need not be 1-to-1. Various options are possible

1. **1-to-0 mapping**
   A **`wpo2`** file is not generated, the original **`wpo1`** file is either renamed with a **`.o`** extension, or used directly in the final link.

2. **1-to-1 mapping**
   A **`wpo2`** file contains no object code nor IR. It only contains the local call graph with the updated summary information. This is used to optimize the IR in the **`wpo1`** file to generate the corresponding **`.o`** file used in the final link.

3. **many-to-1/many-to-many/1-to-many mappings**

   One or more `wpo1` files are combined into one or more `wpo2` files which are then optimized and used to generate the `.o` file used in the final link. This mode of operation may be used when the global call-graph can be partitioned so that functions that are closely related in the program are optimized as a single CU. When this happens, all the functions in the aggregated CU can be optimized with all the IPA passes that require function bodies to operate. Optimizations across CUs will mostly use summary-based transformations.

This stage may implement an incremental approach using the global summary database. This document does not define what the database should contain, nor how WPA may use its contents. Conceptually, this database contains all the results from global analysis collected during the previous compilation of the application. This enables WPA to process only those `wpo1` files that have changed (e.g., by using checksums on the GIMPLE bodies).

The result of WPA is a local transformation plan that is emitted into every `wpo2` file. Also, in the case of incremental analysis, the summary database is updated with analysis results and any other state required to make incremental decisions later. Furthermore, for incremental compilations, if neither the transformation plan nor the input `wpo1` files have changed according to checksum information in the summary database, then the corresponding `wpo2` files need not be generated.

WPA may decide to partition the call-graph. In this case, summary information is used to assign weights to the nodes and edges of the call-graph. Edge weights approximate the cost of missed optimization opportunities should they cross a partition boundary. Node weights estimate the memory consumed by a node. The call-graph is partitioned so as to limit the sum of edge weights crossing partition boundaries while evenly distributing node weights. All the functions in the same partition are emitted to a single `wpo2` file. In this cases, functions from multiple input files will be compiled as if they came from a single source file. This enables the local transformation phase (LTRANS) to perform inter-procedural optimizations that require the presence of every function body.

During analysis, WPA will likely find edges in the global call-graph that go into third party code for which no source code is available. This is typical of external and system libraries. At these boundaries, the default behaviour is to assume the worst and consider the edge a clobbering site or escape point. However, if a summary database is available for the third party code, WPA should use it to its advantage. This summary need not be overly complex, it may simply be mod/ref information for specific kinds of symbols or types. Note that the implementation of such a system needs to be careful with respect to versioning. The information in the summary database must match the library being linked in. Additionally, for dynamically linked code, it may not even be safe to use the summary database, as the application may load different versions of the same code at different times during execution.

In terms of concurrency, the WPA stage is sequential. While it may be possible to multi-thread some or most of its activities, we do not expect this phase to exhibit good concurrency, nor we expect this phase to be amenable to distribution. Even a multi-threaded WPA will require several synchronization points to access the incremental database, build the global call-graph, perform transitive closures, etc.

Local Transformation (LTRANS)

The compiler is invoked again on each **`wpo2`** (or **`wpo1`**) file for final optimization. All the transformation decisions and analysis results from WPA are implemented locally in each file. This generates final **`.o`** files that are sent to the linker to generate the target executable.

Each **`wpo2`** file represents a subgraph of the global call-graph for the application. It may even contain duplicate call-graph nodes to satisfy the requirements of certain transformations like inlining. For instance, suppose that the partitioning decided during WPA specifies that function **`foo()`** is to be inlined inside two different partitions **`f1.wpo2`** and **`f2.wpo2`**. Since both **`wpo2`** sets are going to be handled by different LTRANS nodes, the body of **`foo()`** needs to be replicated in both **`f1`** and **`f2`** partitions.

Memory is an important consideration during LTRANS. Either the size of **`wpo2`** files must be strictly controlled by the WPA partitioning scheme, or LTRANS must not require all function bodies to be in memory at the same time. In the latter case, a crucial implementation feature will be the ability to read GIMPLE bodies on demand very quickly.

Similar to LGEN, this phase is highly concurrent. It can be multi-threaded on multiprocessors and also distributed on clusters. Every **`wpo2`** file contains a transformation plan that applies to all the functions in the compilation unit.

Besides the transformation plan, each **`wpo2`** file will contain summary results from WPA that apply to all the local functions. This may include the global call graph or just summary information for every call-graph node referenced by the compilation unit.

Potential transformations

We distinguish two classes of IPA transformations that are supported by this approach

Global decisions with local transformations

These transformations only need summary information on each call-graph node to drive optimization decisions during the optimization of `wpo2` files. These are the ideal transformations from the point of view of scalability, since they are highly parallelizable and do not require significant amounts of IL to be live in memory simultaneously.

The following are some transformations that are possible to perform using this approach:

1. Devirtualization
2. Inline caching
3. PIC optimizations
4. Common block padding and splitting, array padding
5. Global variable analysis (assigned ones, global->static)
6. Class hierarchy analysis, static cast removal
7. Indirect call promotion
8. Dead variable elimination
9. Dead function elimination
10. Field reordering, structure splitting/peeling
11. Data reuse analysis
12. Inter-procedural prefetch

Global decisions with global transformations

These transformations need to hold each function body in memory or at least be able to load function bodies on demand. Consequently, these transformations are more expensive. LTRANS may perform these transformations, but without the benefit of a global view. Their effectiveness will depend on how well the global call-graph is partitioned during WPA. Some of these transformations may also be modified to operate only on summary information.

1. Cross-module inlining
2. Virtual function inlining
3. Cloning
4. Inter-procedural points-to
5. Inter-procedural constant propagation

Summary Information

The results of whole program analysis produce information in three scopes

1. The call-graph
2. A call-graph node
3. A call-graph edge

Each new analysis/optimization will add fields to each data structure but special care must be taken with information stored in nodes and edges. Currently, on a 64-bit host, each call-graph node uses 216 bytes, and each call-graph edge uses 88 bytes. This means that a call-graph with 1M nodes and 1M edges will require a minimum of 304Mb of storage. Therefore, we think that call-graph nodes should not grow much more beyond 1,000 bytes or so.

Implementation Plan

Finish IR read/write

The `lto` branch can currently write-out and read-in GIMPLE bodies in several test cases. However, this is still far from complete:

1. The reader loads in memory every function body of every .o file it receives.
2. Call-graph analysis and optimization assumes that all the function bodies are available and loaded in memory.
3. Basic reader/writer implementation is not complete and is likely inefficient.

Tuple representation for GIMPLE

This work is being done in the `gimple-tuples-branch` branch. The conversion to a tuple data structure will allow for more efficient reader and writer stages as well as improved memory utilization. Currently, the branch is emitting RTL from tuples but none of the optimization passes have been converted. This work can continue in parallel with the other tasks on the `lto` branch, but it will force a reimplementation of the reader/writer stages when finished.

Eliminate language hooks

Language hooks ("langhooks") are call-back methods in the Front End (FE). This is used when the language-independent parts of the compiler need language-dependent information, such as whether two types are compatible. Information that used to be taken from the FE before should now be either not needed because it is encoded directly in the IL, or made independent from the FE by defining support in the middle end (e.g., the definition of a type system for GIMPLE).

IPA<->source type unification

One of the consequences of separating front end processing from global analysis and optimization is that during the analysis phase (WPA), the compiler has lost information about the original source language. Furthermore, type rules that apply to one particular language will generally not apply to another. Notably, this affects type-based aliasing rules, such that given a pointer P1 and an addressable variable G, it is not possible to use language aliasing rules to decide whether P1 may point to G.

One alternative would be to encode the original source language in every type emitted for analysis. However, this does not address the cross-language problem as language standards generally do not specify type-compatibility rules across languages. Initially, we propose to use the idea of alias sets already implemented in GCC so that

1. Two types coming from different languages are always assumed to have the same alias set
2. Two types within the same language, use the alias set encoding defined by their original language

This arrangement should allow the global optimizer to use traditional type-based alias rules when analyzing types originating from the same language and be conservative when mixing two different languages. A more aggressive alternative could be implemented by having each front end language define additional attributes. For instance, if structural equivalence rules are allowed by a language, this attribute could be enabled during parsing and used when computing type-based alias analysis during global optimization.

To summarize, we propose adding attributes on each **`TREE_TYPE`** created by the front end to identify the language for that type. Additionally, if the language allows structural equivalence to be used in type compatibility tests, it should be marked as a type attribute.