Bypass assembler when generating LTO object files

Abstract:

Link Time Optimization (LTO) enables GCC to dump its internal representation (GIMPLE) to disk so that a single executable can be optimized as a single module. LTO is implemented as a GCC frontend for a bytecode representation of GIMPLE emitted in special object file sections. Currently, LTO support is enabled in most ELF-based systems, as well as Darwin, Cygwin, and MinGW systems. The current implementation creates an assembly file, and the assembler is used to create the final LTO object file.

This project aims to create Link-time-optimization (LTO) object files directly from the compiler to improve compile time performance significantly by bypassing the assembler. The GCC's LTO infrastructure has matured enough to compile large real-world applications. Thus, this project will significantly reduce their compile time once completed.

Current State of Project:

The current implementation of GCC writes the IL representation along with other sections, such as debug info, symbol table, etc., in an assembly file. Then, the assembler is used to convert it into LTO object files. Streaming of individual sections is implemented in lto-streamer-out.cc, which can either be used to produce assembly code containing the section data (dumped hexadecimally, this is done currently) or simple-object API provided by libiberty to produce object files directly (the aim of this project).

The current layout of the LTO object file can be found here: https://gcc.gnu.org/onlinedocs/gccint/LTO-object-file-layout.html#LTO-object-file-layout. In the slim object file (Default when using -flto, fat lto can be obtained using -ffat-lto-object), some sections contain the IL and other contains the info related to debugging, command line options, symbol table, etc. Using an assembler for this is clearly an overhead as we aren't saving GIMPLE bytecode.

Jan Hubicka created a patch (It was a proof of concept and not fully tested) that directly outputs the object file from the compiler using the libiberty simple-object.c API. However, since simple-object.c is unable to create a complete object file from scratch, an existing object file was required to fetch the target's file attribute.

The patch still requires work to be completed.

1) Passing existing object files isn't optimal and simple-object API should be extended to produce an object file from scratch.
2) The current way to run and compile, as described in Jan's patch is to first stop the compiler before the assembling step (by passing the -S option and -o <object-flie>.o) and then invoke gcc again to produce the executable.
3) Pach is completely missing handling of dwarf debug information and crashes when the -g option is passed.

Project Goals:

The goal of this project is to reduce compilation time while using LTO mode.
These are results on the patch as tested by Jan (I am directly copy-pasting)

For 1000 invocations with bypass:

```
real    0m14.186s
user    0m10.957s
sys     0m2.424s
```

While the default path gets:

```
real    0m21.913s
user    0m13.856s
sys     0m5.705s
```

With OpenSUSE 13.1 default GCC 4.8.3 build:

```
real   0m15.160s
user   0m8.481s
sys    0m5.159s
```

And with clang-3.4:

```
real   0m30.097s
user   0m22.012s
sys    0m6.649s
```

I also tested the patch on the current source and got this.

Bypass:

```
real    0m18.956s
user    0m11.889s
sys    0m6.921s
```

default:

```
real    0m27.531s
user    0m17.560s
sys     0m9.729s
```

The speedup in compile time is noticeable. As GCC's LTO infrastructure has matured enough to compile large applications such as LibreOffice and Firefox, this project will significantly reduce their compile time if completed.

Implementation:

Here is how I plan to complete the patch by Jan Hubicka (This may be changed/improved if I find some better way to do the same in the investigation/discussion phase).

1) I plan to extend simple-object.c to output missing information such as architecture info and symbol table, and to handle output symbols __gnu_lto_slim as COMMON. This will enable simple-object.c to create an object file from scratch.
2) Modifying the gcc driver. This will be necessary as in the current scenario we need a workaround to produce an executable from the patch (See pending work point 3 in the Current state of the project section).
3) Handling the output of dwarf debug information. The current patch doesn't handle debugging information output. It simply crashes(Segmentation fault) when passing the -g option. This is necessary because debugging is supported in LTO mode.

NOTE: As suggested by Jan Hubicka (seems reasonable to me too) I will first implement ELF path first to get a work ing implementation and add support for other object formats incrementally.

Timeline Of the Project:

My college final exams end on April 28 and I don't have any internship/projects during this summer so I will be working full-time with GCC if selected. Thus I will be able to devote 7-8 hours per day (40+ hours per week) to this project.
I have already built and tested GCC and have an understanding of simple-object.c lto-object.c and lto-streamer so I don't need to devote a lot of time to understanding the code. Although I have read the contribution guidelines and David's newbie guide, I will be spending my free time in April to get a good grasp on those aspects.

I plan to complete this project in four phases
Here is the detailed timeline:
Phase 0 (Pre GSOC period till May 4):

1) Get familiar with xcoff, mach and coff handler of simple-object(I am already familiar with

elf)
2) Read more about testing GCC, Contribution guidelines, making a patch etc.

Phase 1 :
1) Understanding the codebase in more detail
2) Extending the simple-object.c to output the missing information
3) Properly configure the driver to produce executables directly when passing the -fbypass-asm option

This will require roughly 2-3 weeks of time.

Phase 2:
1) Test the code written in the previous phase on GCC test suite.
2) Read the documentation of dwarf-debug in more detail

This will require roughly 1-2 weeks of time.

Phase 3:
1) Handle the output of dwarf debug information

This will require roughly 2-3 weeks of time.

Phase 4:
1) Finally test all implementations on a comprehensive GCC test suite.
2) Writing documentation as per need.

This will require roughly 1-2 weeks of time. (Testing won't require a lot of time but debugging will :))
I have kept a buffer of approximately 2-3 weeks if things go south or maybe debugging some specific portion takes a lot of time.

About Me:

- Name - Rishi Raj
- University - Indian Institute of Technology Kharagpur
- Email - rishiraj45035@gmail.com
- GitHub username: rsh-raj
- Time Zone - IST (GMT + 05:30)
- Prefered Language for communication: English

I am a third-year undergraduate student pursuing a dual degree in the Department of Computer Science & Engineering at the Indian Institute of Technology Kharagpur. Since my first year in 2020, I have been programming in C/C++. I wish to work with GCC this summer as a GSOC student and beyond. I believe I have the necessary skills to undertake this project.

Over the last three years, I have completed several programming projects and assignments in C/C++, most of which are available in my public github repository. Additionally, I have previously interned at the Indian startup Zylu, where I primarily worked with PHP. During my internship, I gained experience in navigating large codebases, comprehending them, and modifying them to implement new features.

I have taken compiler theory and laboratory courses as part of my institute curriculum. In the laboratory, we designed a tiny-c compiler (a subset of GCC, GitHub link: click here). In theory, I learned about different phases of compilations, various optimization techniques, etc. Please click here to go to my course website for a detailed overview. This course sparked my interest in compiler development, and I am eager to take it further by making meaningful contributions to GCC. I hope to make some significant contributions to GCC this summer and in the future.

I have also taken a Computer Architecture and Operating System course, so I am well-acquainted with low-level code and the ELF file format. Currently, my Operating System assignment deals exclusively with ELF files. I plan to make it public on my Github by April 15th, which is the submission deadline.

My experience with GCC:

I have been using GCC for three years now. After taking a compiler course I got intrigued to examine the internal working of GCC, that's what brought me here. I started off with David's Newbie guide and got a good understanding of how to build test and debug GCC. Up till now, I have a fair idea of the workings of LTO except for a deeper understanding of various optimization techniques. As I am more inclined toward this project so I decided to investigate the current patch, libiberty/simple-object.c, and lto-streamer more as a result of which I have a good understanding of their codes. I have also read about different contribution guidelines like coding styles, and how to make a patch, and was able to successfully apply Jan Hubicka's patch on the current source.

I really enjoyed learning about internal working LTO and even if I don't get selected I will wish to continue this project this summer. I would also like to mention that the GCC community is very supportive(My experience after reading various replies in mailing lists).

Post GSOC:

As I have mentioned multiple times earlier, I am genuinely interested in compiler development, and I will always keep up with GCC's development and make my best effort to contribute. Furthermore, I will always be available for any future changes or extensions in this project.

References:

I have added links to wherever needed above, these are additional references.

https://gcc.gnu.org/pipermail/gcc/2023-March/240833.html
https://gcc.gnu.org/pipermail/gcc/2022-May/238670.html
Skipping assembler when producing slim LTO files
Google Summer of Code 2023 Timeline
LTO Overview (GNU Compiler Collection (GCC) Internals)

[1010.2196] Optimizing real world applications with GCC Link Time Optimization
https://gcc-newbies-guide.readthedocs.io/en/latest/index.htm
Contributing to GCC - GNU Project
https://gcc.gnu.org/pipermail/gcc/2023-April/241086.html (Whole thread).