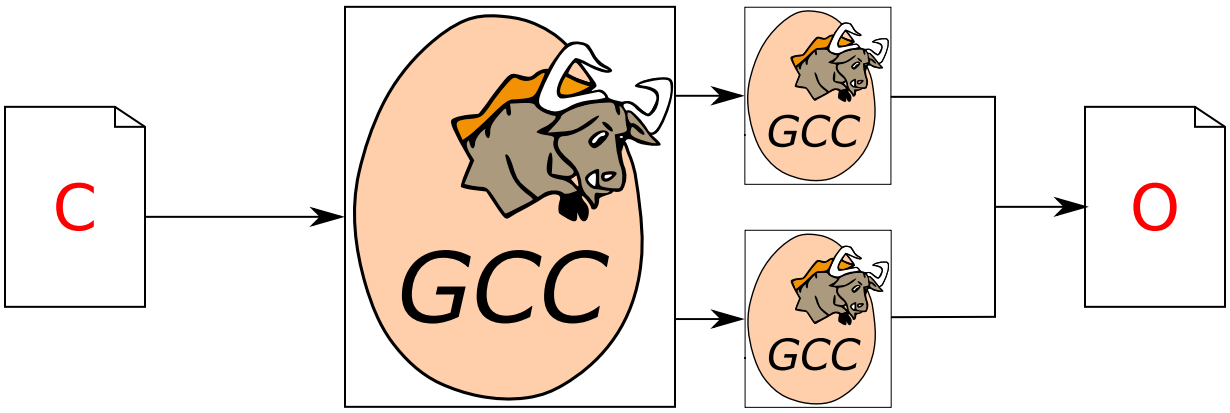
Automatic Detection of Parallel Compilation Viability



Giuliano Belinassi
Timezone: GMT−3:00
University of São Paulo – Brazil
IRC: giulianob in #gcc
Email: giuliano.belinassi@usp.br
Github: https://github.com/giulianobelinassi/
Date: March 24, 2020

1 About Me

Computer Science Bachelor (University of São Paulo), currently pursuing a Masters Degree in Computer Science at the same institution. I've always been fascinated by topics such as High-Performance Computing and Code Optimization, having worked with a parallel implementation of a Boundary Elements Method software in GPU. I am currently conducting research on compiler parallelization and developing the ParallelGcc project, having already presented it in GNU Cauldron 2019.

Skills: Strong knowledge in C, Concurrency, Shared Memory Parallelism, Multithreaded Debugging and other typical programming tools.

2 Brief Introduction

In ParallelGcc, we showed that parallelizing the Intra Procedural optimizations improves speed when compiling huge files by a factor of 1.8x in a 4 cores machine, and also showed that this takes 75% of compilation time.

In this project we plan to use the LTO infrastructure to improve compilation performance in the non-LTO case, with a tradeoff of generating a binary as good as if LTO is disabled. Here, we will automatically detect when a single file will benefit from parallelism, and procceed with the compilation in parallel if so.

3 Use of LTO

The Link Time Optimization (LTO) is a compilation technique that allows the compiler to analyse the program as a whole, instead of analysing and compiling one file at time. Therefore, LTO is able to collect more information about the program and generate a better optimization plan. LTO is divided in three parts:

- *LGEN (Local Generation)*: Each file is translated to GIMPLE. This stage runs sequentially in each file and, therefore, in parallel in the project compilation.
- *WPA (Whole Program Analysis)*: Run the Inter Procedural Analysis (IPA) in the entire program. This state runs serially in the project.
- *LTRANS (Local Transformation)*: Execute all Intra Procedural Optimizations in each partition. This stage runs in parallel.

Since WPA can bottleneck the compilation because it runs serially in the entire project, LTO was designed to produce faster binaries, not to produce binaries fast.

Here, the proposed use of LTO to address this problem is to run the IPA for each Translation Unit (TU), instead in the Whole Program, and automatically detect when to partition the TU into multiple LTRANS to improve compilation performance.

The advantage of this approach is: by partitioning big files into multiple partitions, we can improve the compilation performance by exposing these partitions to the Jobserver. Therefore, it can improve CPU utilization in manycore machines. Generated code quality

should be unaffected by this procedure, which means that it should run as fast as when LTO is disabled.

- It can generate binaries as good as when LTO is disabled.
- It is faster, as we can partition big files into multiple partitions and compile these partitions in parallel.
- It can interact with GNU Make Jobserver, improving CPU utilization.

4 Planned Tasks

I plan to use the GSoC time to develop the following topics:

- Week [1, 4] – May 4 to May 27:
  Update `cc1`, `cc1plus`, `f771`, ..., to partition the Compilation Unit (CU) after IPA analysis directly into multiple LTRANS partitions, instead of generating a temporary GIMPLE file, and to accept a additional parameter `-fsplit-outputs=<tempfile>`, in which the generated ASM filenames will be written to.

  There are two possible cases in which I could work on:

  1. *Fork*: After the CU is partitioned into multiple LTRANS, then `cc1` will fork and compile these partitions, each of them generating a ASM file, and write the generated asm name into `<tempfile>`. Note that if the number of partitions is one, then this part is not necessary.
  2. *Stream LTRANS IR*: After CU is partitionated into multiple LTRANS, then `cc1` will write these partitions into disk so that LTO can read these files and proceed as a standard LTO operation in order to generate a partially linked object file.

  1. Has the advantage of having less overhead than 2., as there is less IO operations, however it may be hard to implement as the assembler file may be already opened before forking, so caution is necessary to make sure that there are a 1 - 1 relationship between assembler file and the compilation process. 2. on the other hand can easily interact with the GNU jobserver.

- Week [5, 8] – June 1 to June 26:
  Update the `gcc` driver to take each file in `<tempfile>`, then assemble and partially link them together. Here, an important optimization is to use a named pipe in `<tempfile>` to avoid having to wait every partition to end its compilation before assembling the files.

- Week 9 – June 29 to July 3: **First Evaluation**
  Deliver a non-optimized version of the project. Some programs ought to be compiled correctly, but probably there will be a huge overhead because so far there is no way of interacting with GNU Jobserver.

- Week [10, 12] – July 6 to July 24:
  Work on GNU Make Jobserver integration. A way of doing this is to adapt the LTO WPA → LTRANS way of interacting with Jobserver. Another way is to make the forked `cc1` consume Jobserver tokens until the compilation finishes, then return the token when done.

- Week 13 – July 27 to 31: **Second Evaluation**
  Deliver a more optimized version of the project. Here we should filter files that would compile fast from files that would require partitioning, and interact with GNU Jobserver. Therefore we should see some speedup.

- Week [14, 16] – August 3 to 21:
  Develop adequate tests coverage and address unexpected issues so that this feature can be merged to trunk for the next GCC release.

- Week 17: August 24 to 31 **Final evaluation**
  Deliver the final product as a series of patches for trunk.