Compiling Files in Parallel: A Study with GCC

Giuliano Belinassi
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, Brazil
giuliano.belinassi@usp.br

Richard Biener
SUSE Labs
Nuremberg, Germany
rguenther@suse.com

Jan Hubička
Faculty of Mathematics and Physics
Charles University
Prague, Czech Republic
hubicka@kam.mff.cuni.cz

Daniel Cordeiro
School of Arts, Sciences and Humanities
University of São Paulo
São Paulo, Brazil
daniel.cordeiro@usp.br

Alfredo Goldman
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, Brazil
gold@ime.usp.br

***Abstract*—Processors are becoming increasingly parallel, but compiling software has so far been a task parallelizable only by the number of files in it. To improve compilation parallelism granularity, we propose a method feasible to implement in commercial compilers for single file parallel compilation, with simple modifications in the Link Time Optimization (LTO) engine; which we show by implementing it in GCC. This method resulted in a 35% speedup when self-compiling GCC when compared to `make -j` only parallelism, and up to $3.5\times$ speedup when compiling individual files. We also explain why the adoption of our proposed method is still compatible with the Reproducible Builds project.**

***Index Terms*—Compilers, Parallel Compilation, Link Time Optimization, LTO**

I. Introduction

The recent advances in both technological and computational fields induced an increasingly faster expansion of software ecosystems. Developers create new programs to supply the needs of the most diverse domains, either through web systems coded in script languages; or by components to an operating system destined to control some hardware resources. Regardless of the reason behind their development, it is true that their code will be, at some point, transformed into machine language by a compiler or assembler, even on interpreted languages due to Just In Time (JIT) compilers often used in them.

Compilers are enormous programs developed and enhanced by people in both industry and academia. A compiler is nothing more than software that translates code from a programming language A (the source language) into another language B (the target language) [1]. This translation permits enhancing the code written in the source language (usually written by a human). It permits, for instance, error detection in the code written in the target language or performance optimizations of the generated code.

There are several studies about compilers' improvement at generating more efficient code without giving away correctness. We can cite, as examples, two well-known open-source projects with a vast community of developers and users that share this goal: the GNU Compiler Collection (GCC)[1] and LLVM[2], which are capable of translating several languages such as C, C++, and Fortran, to machine language.

GCC was initiated by Richard Stallman, with its first public release in March of 1987, supporting the C language and targeting several architectures [2]. Today, GCC is a global collaboration project, with hundreds of thousands of lines of code and perhaps the most used C/C++ compiler in the Linux ecosystem.

GCC was initially designed to compile programs one file at a time, meaning that it could not allow global cross-file optimizations because the compiler never has the opportunity to analyze the program as a whole. New optimization opportunities are now possible after *Link Time Optimization* (LTO) was proposed [3], [4] and implemented [5].

LTO (see Section III-A) is a method for interprocedural optimization that is performed at the time of linking the application code. With LTO, at linking time an optimization algorithm can use dependency information from all source files, even if they were compiled separately. GCC supports LTO by using the `-flto` flag. Part of the LTO engine had already been parallelized, which inspired this study.

The main contribution of this research[3] is a parallelization method to compile single files and its implementation in GCC, an industrial-scale compiler. We address that by reusing the pre-existing GCC's LTO engine for partitioning the single file Translation Unit (TU) after the Interprocedural Analysis (IPA) was decided, and we proceed to compile each partition individually, in parallel. IPA includes only interprocedural optimizations, which require interactions among distinct functions to optimize (*e.g.* inliner). There is already a way to instruct the LTO engine to partition a single file TU, by using `-flto -flinker-output=nolto-rel`, but with phony

[1]https://gcc.gnu.org/
[2]https://llvm.org/
[3]Supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, Google Summer of Code, FAPESP proc. 2019/26702-8, and FAPESP proc. 2021/06867-2.

978-1-6654-5157-4/22/$31.00 ©2022 IEEE
DOI 10.1109/SBAC-PADW56527.2022.00011

object creation overhead and no guarantee of correct name clash resolution, as discussed in Section IV-D.

The remaining of this paper is organized as follows. Section II introduces the GNU Compiler Collection and presents relevant internal details about its implementation. Previous efforts in compiling a single file in parallel, as well as an introduction to LTO, are presented in Section III. A new parallelization scheme for GCC's LTO is presented Section IV, while its correctness is studied in Section V. Finally, we show our results in Section VI and discuss how to further improve these results in Section VII.

II. The GCC Compiler

The GNU Compiler Collection—also known as GCC—supports several source languages, such as C, C++, Fortran, Go, Ada; and several architectures, such as i386, amd64, riscv, arm, ppc, and pdp-11. GCC is also an optimizer compiler, which means it can modify the input code to generate a more efficient version without changing the program semantics. This is archived using different intermediate representations (IR): GENERIC, GIMPLE, and RTL. The compiler runs optimization passes on each of these IR.

A. GENERIC

GENERIC is an IR that allows representing any function in the program with trees. Such language is a superset of all functionalities and attributes that every source language has—for instance, attributes such as `volatile` in C, C++, or even Fortran matrix sum or product. Other functionalities are more common in every language, such as function calls, variable declaration, class, expressions, statements, and others [6].

This language contains a faithful representation of the program closest to what was written by the programmer, meaning that there are no modifications in the program. GENERIC offers an easy translation to GIMPLE, the hardware-independent IR of GCC in which most optimizations are done, and also simplifies the front-end development. However, it does not mean that optimizations can not be applied here.

There is a match-and-replace mechanism in GCC supporting GENERIC. This mechanism is very simple: it searches for a chain of statements in the IR for a pattern and then replaces it for another. Such patterns are expressed in the file `gcc/match.pd`. This simple mechanism enables several bitwise logic and mathematical simplifications to accelerate the code significantly in some particular cases [7].

Finally, each front-end of GCC is responsible for translating the source code into this intermediate language. To visualize a program in this representation, one compiles the program with the flag `-fdump-tree-original`. However, the documentation shows clearly that C and C++ front-ends translate directly into GIMPLE [8].

B. GIMPLE

GIMPLE is another IR from GCC, fundamentally consisting of conversion from GENERIC to a three-address code representation and having a further step into lowering this representation to construct the control-flow graph and its static single form (SSA). As GENERIC, GIMPLE is general enough to represent every attribute from all source languages supported by the compiler. This representation is based on SIMPLE, an IR from the McCAT compiler [8].

It is in GIMPLE that all hardware-independent optimizations are performed. GIMPLE has an *Application Programming Interface* (API) for interacting with statements, tree nodes, the control-flow graph, SSA nodes, and the call graph. GIMPLE has two states: the "high" state, with lexical scopes and nested expressions, and the "low" state, exposing all implicit jumps and Exception Handles regions, removing the lexical scopes, merging similar return statements, and converting try-catch control flows to exception regions.

GIMPLE code can be dumped by using the flag `-fdump-tree-gimple`, or even `-fdump-tree-ssa` to write the SSA representation on the disk.

C. Register Transfer Language

The last IR of GCC is the *Register Transfer Language* (RTL). The syntax of this language resembles Lisp, in which expressions are represented by a list of commands [9].

RTL is a language that aims to be as close to the assembler language as possible while still being generic enough to be compatible with every target language of GCC. It represents a machine with an infinite number of registers that will be correctly selected to a machine register on the register allocation pass, or allocated in memory.

The final step is the assembler emission once all optimizations were performed in RTL. IT will employ a macro translation instruction selection algorithm to convert RTL functions, statements, and expressions into assembler. That is why RTL optimizers are necessary here to avoid generating poor-quality assembler code [10].

D. Optimization Passes

GCC has several optimization passes that are applied to each intermediary language. There are three types of passes:

1) **IPA** — an interprocedural pass. This pass operates on the call graph of the program being compiled. It often has three step passes: on the first run, it collects a *summary* of the function, then it will *analyze* the call graph with this summary (for instance, to determine if inlining a certain function will not blow the code's size); finally it will *transform* the code. The reason behind it is that the *analysis* phase must be able to run without having the entire function loaded in memory;
2) **GIMPLE** — a hardware-independent intraprocedural pass. It can analyze the code (such as collecting some metrics) or even do an extensive transformation in the program. One example is the `pass_vectorize`, which tries to find vectorizable operations in the code and operates on GIMPLE;
3) **RTL** — a hardware-dependent intraprocedural pass, operating on RTL.

III. Related Works

Parallel Compilation includes parsing (parallel or not), how to perform analysis, optimization, and code translation in parallel. Parsing can be described as building a machine to decide if an input string is a member of a stated language or not, creating the Abstract Syntax Tree in the process by logging the used derivation rules.

Parallel Parsing dates back to 1970. Lincoln [11] explored how to use the vectorial registers in the (so far) STAR-100 supercomputer for lexical analysis. Fischer [12] gives a detailed theoretical study, proving several concurrent parsing techniques for the LR(k) family. The parser proceeds by breaking the input into several arbitrary parts and running a serial parser on each of them. Then, the algorithm tries to recover the stack of each noninitial parser by constructing a set of possible states, for which there are five potential cases. However, in case of an error, the parser result should be discarded, and therefore much work will be done in vain when compared to the sequential version.

Fowler and Joshua [13] described how to parallelize Earley and Packrat methods. For the former, the authors partition the Earley sets into sub-blocks and run each block in parallel. To solve the dependency across the blocks, the authors propose a way to speculate additional items into the parser, which are not produced by the serial algorithm. For Packrat, the authors propose a message passing mechanism. The input is divided into parts, and each part is assigned to a worker thread. Each thread speculates until the thread on its left has finished parsing, and send the synthesized starting symbol to the other thread on its right. The authors managed a speedup of $5.5\times$ in Earley and $2.5\times$ in Packrat.

Also regarding Packrat methods, Dubroy and Warth [14] show how to implement an incremental version of Pacrkat to avoid reparsing the entire input on modifications. The authors achieve this by recording all its intermediate results in a parsing table and modifying the parsing program to reload this table when invoked. They showed that their method does not require any modification to the original grammar, and their technique only requires up to 11.7% extra memory compared to the original parser. Besides, the authors claim a reduction from $23.7ms$ to $6.2ms$ on reparsing a 279Kb input string.

Barenghi *et al.* [15] explore some properties of Operator Precedence Grammars to construct a Yacc-like parser constructor named PAPAGENO, which generates parallel parsers. The authors described precedence grammars for Lua and JSON, which they used in their tests to get a speedup of up to $5.5\times$ when compared to a parser generated by GNU Bison.

Regarding parallel compilation, the literature dates back to 1988. Vandevoorde [16] worked on a C compiler, and Wortman and Junkin [17] worked on a Modula-2+ compiler. The former assumes that every function declaration is in the file headers and implements per-function and per-statement parallel compilation. The latter implements only per-function parallelism. Speedups ranged from $1.5\times$ to $6\times$ on a multicore MicroVAX II machine. None of these papers discuss optimization, and they concentrate on (today's perspective) non-optimizer compilers, which is not the case with GCC.

Lattner *et al.* proposed MLIR, an *Intermediate Representation* (IR) that aims to unify several Machine Learning frameworks and compilers IR [18]. Its design also includes support for multithreaded compilation by admitting concurrent transversal and modification of the IR.

There has been an attempt of parallelizing GCC by threading the GIMPLE intraprocedural pass manager [19], which only requires information contained inside the function's body (*e.g.* vectorization). The authors managed a speedup of up to $3.35\times$ to this compilation stage, and up to $1.88\times$ in the total compilation time of a file when extending this technique to the RTL passes.

A. Link Time Optimization (LTO)

Compilation usually uses the following scheme: a compiler consumes a source file, generating an assembly file as output. This file is then assembled into an object file and later linked with the remaining objects file to compose an executable or a library. Figure 1 illustrates this process for a single file. In this paper, we name this method the *classical compilation* scheme.

The problem with this approach is that it can only optimize with the information found in its Translation Unit (TU) due to its inability to get to the body content of other files' functions. A TU is composed of the entire content of a source file (*e.g.*, a `.c` file in C) plus all its headers.

In order to solve that issue, LTO admits cross-module optimizations by postponing optimizations and final translation to a linker wrapper. There, the entire program can be loaded by the compiler (more often, just some sort of summary) as a single, big TU, and optimizations can be decided globally since it has access to the internals of other modules. LTO is divided into three steps [4], [5]:

- LGEN (*Local Generation*): each module is translated to an intermediate representation and written to disk in phony object files. These objects are *phony* because they do not contain assembly code. This step runs serially on the input file (*i.e.* in parallel to the files in the project).
- WPA (*Whole Program Analysis*): loads all translated modules, merges all TUs into one, and analyzes the program globally. After that, it generates an *optimization summary* for the program and partitions this global TU for the next stage. This analysis runs sequentially to the entire project.
- LTRANS (*Local Transformations*): apply the transformations generated by WPA to each partition, which in turn will generate its own object file. This stage runs in parallel.

This process is depicted in Figure 2, where the linker wrapper is represented by *collect2*, which firstly launches *lto1* in WPA mode, and then finally launches *ld*. This process can be observed by launching gcc with `-flto -v`.

Glek and Hubička [5] notice that LTO produced faster and smaller binaries when compiling GCC and Firefox even

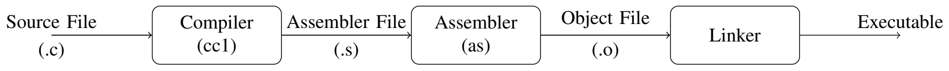

Fig. 1. GCC Compiling a `.c` file in *classical compilation* mode

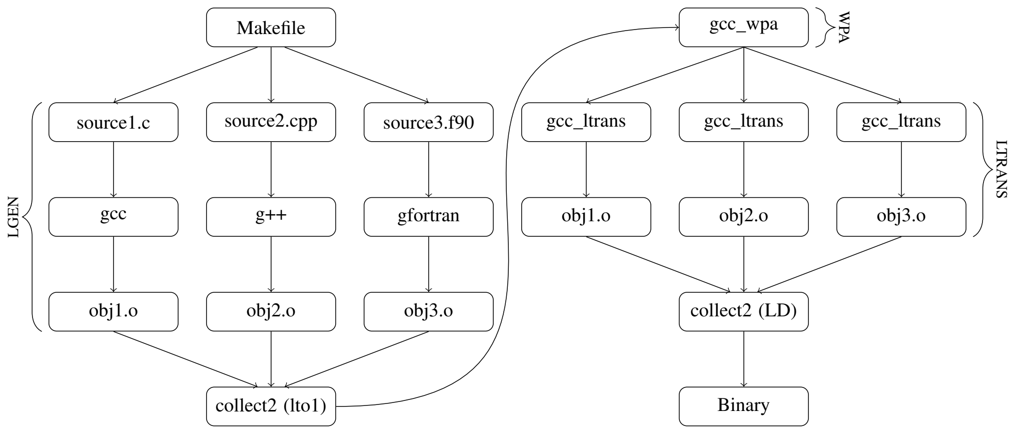

Fig. 2. Compilation of a program using LTO scheme

though with a two-time slower compilation in GCC. Interestingly, Firefox compiled 20 minutes faster. In all reported cases, the intraprocedural optimizations dominated the compilation time.

Liška [20] complements the analysis of this functionality. One of the problems found was the high memory consumption of LTO, which went up to 12 GB when compiling Chromium and 15 GB when compiling Linux. The major complaint was the necessary recompilation time when a single file is modified in LTO mode compared to the classical compilation scheme.

Furthermore, there is research using a profile-based approach rather than a whole-program analysis for cross-module optimizations. Xinliang *et al.* [21] proposed LIPO, which embeds IPA in the generated binary with minimal overhead, and loads information about functions in other modules based on the profiling report. LIPO also removes the requirement of phony object dumps and link-time wrappers in opposition to LTO. Results showed a speedup of up to 10$\times$ in compilation time compared to Feedback Driven Optimization (FDO) alone.

Johnson *et al.* propose ThinLTO [22]. It differs from LTO because it generates the function summaries on the LGEN stage for improved parallelism, avoiding the large memory requirements of the original LTO. ThinLTO supports incremental builds by hashing the compiled function to avoid unmodified functions recompilation, while LTO requires recompilation of the entire partition. Most GCC's IPA optimizations only use summaries to prevent this memory footprint issue.

In our study, we present a way to use the LTO engine to compile single files in parallel, in contrast to the previously discussed papers which developed approaches to analyze the program as a whole for better optimizations, without the intent to compile faster. As for optimization quality, our method results in optimizations as good as the classical compilation scheme.

IV. A Parallel Approach for GCC's Link Time Optimization

As presented in Section III-A, LTO was created to allow cross-module optimizations in programs and has a serial part (WPA), imposing a compilation bottleneck on many-core machines. Furthermore, the *classical compilation* scheme cannot partition its Translation Unit (TU) for parallel compilation, which can also be a bottleneck for the process if it is too large. The latter issue, however, can be solved by transplanting the LTO partitioner to the *classical compilation* scheme and making it work *without* having the context of the *entire* program. We show that this transplant is possible by implementing it in GCC.

Our approach differs from LTO in how we handle the IPA. LTO manages these optimizations with the whole program context, while our method will only have the context of the original TU. It allows optimizations as good as they are in the *classical compilation* scheme while benefiting from the extra parallelism opportunity available in the LTO's LTRANS stage.

In this section, we present an important piece of the compiler from the User Experience perspective: the `gcc` *driver*. Then, in Section IV-B, we explain a short algorithm for making the LTO partitioner work for our proposal. In Section IV-C we present a necessary change we had to make in our work about how partitions are applied in GCC. In Section IV-D, we explain how we solved the issue of private symbols with the same name being promoted to global. Subsequently, in Section IV-E, we present our work about communicating with the GNU Make Jobserver to keep track of used processors. Finally, we show why our proposal is still compatible with the Reproducible Builds project in Section IV-F.

A. The `gcc` Driver

From a compiler theory perspective, a compiler translates a program from a language A to another language B [1]. In GCC, it translates several languages to assembly for some architecture (*e.g.* x86). This means that encapsulating code in object files, or linking these files in an executable, are not tasks of the compiler itself.

GCC is actually composed of *driver* programs (`gcc`, `g++`, `gfortran`, etc.) that call the compiler itself (`cc1`, `cc1plus`, `f951`, etc.), the assembler (`as`), and the linker (`ld` or `collect2`). There is a compiler for each language.

This explains why a user can, for instance, simply launch `gcc -o binary file.c` and get a working binary. The `gcc` driver will launch the necessary programs for the user to build the binary file. The command line of the previous example launches three programs, as illustrated in Figure 1.

Users rely on the standard behavior of drivers, and a parallel version of GCC cannot change how files are created during the execution of the driver. Therefore, if we want our changes not to break the building scripts (*e.g.*, Makefile) used by most projects—which rely mostly on launching `gcc file.c -c` to create an object file `file.o`—we must ensure that a single object file for each file is created, instead of multiple ones, as does the LTO partitioner. Fortunately, we can rely on GNU *ld* partial linking for merging object files into one. Therefore, the solution to this problem is:

1) Patching the LTO *partitioner* to communicate the location of each created assembly file (`.s`) to the *driver*. This can be made by passing a hidden flag `-fadditional-asm=<file>` by the driver to the partitioner, which the last will write to. Also, it is possible to replace this file with a Named Pipe for better performance if needed.
   Then, the partitioner checks if this flag has been passed to the compiler. If it was, a *compatible version* of the driver is installed. If the partitioner decides to partition the TU, it should *retarget* the destination assembly file and write the retargeted name to the communication file.
2) Patching the driver to pass this hidden flag to the *partitioner* and to check if this file exists. If not, this means that either the compiler is incompatible or has chosen not to partition the TU. In the first case, the driver should call the assembler for every assembly file generated and the linker to generate the expected final object file. In the second case, simply fall back to the previous compilation logic.

Figure 3 illustrates the code flow after these changes. The execution starts in the highlighted node *gcc*, which calls the compiler (`cc1`) with the necessary flags to establish a communication channel among the parts. The compiler then will partition the TU and forks itself into several child processes, one for each partition.

Once multiple processes are created, the compiler will communicate its output `.s` file, the driver will then launch the assembler (`as`, the portable GNU assembler) to generate several object files, and launch the linker (*ld*, the GNU linker) to merge them all into a single object file.

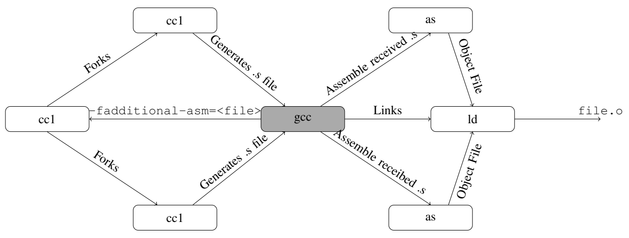

Fig. 3. Interaction between the *driver*, the *compiler*, and other tools after our changes

B. Adapting the LTO Partitioner

In GCC, a Translation Unit (TU) is represented as a call graph. Every function, global variable, or clone (which may represent a function to be inlined) is represented as nodes in a call graph. If there is a function call from f to g, or there is a reference to a global variable g in f, consequently there is an edge from f to g. This means that TU partitioning can be represented as a *graph partitioning* problem.

When LTO is enabled and GCC is in the WPA stage, the body of these nodes is not present, just a *summary* of them (*e.g.* the size in lines of code it had). This is done to save memory. However, when LTO is disabled, the function body is present, resulting in some assertion failures, which were fixed after some debugging.

Afterward, comes the partitioner algorithm *de facto*. In LTO, GCC tries to create partitions of similar size and always tries to keep nodes together. The heuristic used by the original partitioner runs in linear time and its code is quite complex. We decided to design a new partitioning algorithm for this project.

This partitioning algorithm works as follows: for each node, we check if it is a member of a COMDAT [23] group (section where a same symbol is defined by multiple TUs), and merge it into the same partition. After that, we propagate outside of this COMDAT group, checking for every node that may trigger the COMDAT to be copied into other partitions and cluster them into the same partition. In practice, this means to include every node hit by a Depth-First Search (DFS) from the group to a non-cloned node outside the group. Figure 4 represents a sketch of this process.

Initially, we also did this process for private functions, in order to avoid promoting them to public, once external access would be necessary if they go into distinct partitions. However, results showed that this has a strong negative effect on any parallelism opportunity.

For grouping the nodes together, we used a Union-Find with Path Compression algorithm, which yields an attractive computational complexity of $O(E + N \lg^* N)$ to our partitioner, where N is the number of nodes and E is the number of edges in the call graph [24].

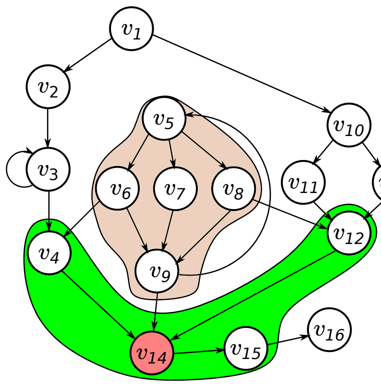

Fig. 4. Example of a call graph, in beige being represented the COMDAT group, in green is the COMDAT frontier and in red are the cloned nodes.

Once the partitions are computed, we need to compute its *boundary*. If function f calls g, but they were assigned into distinct partitions, we must include a version of g in f's partitions without its body, then check if g is a private function. If yes, then g must be promoted to a public function. There is also extra complexity if a version of g is marked to be inlined in f, which means that its body has to be streamed somehow. Fortunately, most of this logic is already present in LTO and we could reuse them.

However, a few issues were found when handling inline functions and global variables marked as part of the boundary. First, some functions marked to be inlined into functions inside the partition were incorrectly marked to be removed. We fixed that by checking if a function planned to be removed is inlined into a function inside the boundary; secondly, variables marked as in the boundary (and therefore, not in the partition) were not being correctly promoted to external. We further fixed that by checking if there were references to that variable through other partitions, and, in this case, we promoted them to global.

Furthermore, there were some issues concerning how GCC handles partitions, which we discuss in the next section.

C. Applying a Partition Mask

Once partitions are computed, the only presented way to apply it (*i.e.*, remove every unnecessary node from the TU) was to reload the compiler and let it load the phony object files. The issue is that these objects are unavailable in our project because we are not running in LTO mode. We developed another method for this. We used the Unix *fork* function to spawn a child process, and then we implemented our own method to apply the partition without having to load these phony object files. Fortunately, this consisted of four cases:

- *Node is in partition*: nothing has to be done.
- *Node is in boundary, but keeps its body*: we indicate that this function is available in other partitions, but we do not release its body or data structures.
- *Node is in boundary*: we mark this mode as having its body removed, but we never actually remove it. This is because the node may share the body contents with another node in the partition. We then remove every edge from its functions to its callees, all references to variables, the content of the dominator tree, and its control-flow graph. This is now a function that only this partition knows about its existence.
- *Node not in boundary*: we remove the node and all its content.

Then, it retargets the output assembly file to another file private to this partition and writes a pair (*partition number, path to file*) into the communication file. The partition number is essential to guarantee that the build is reproducible, as we will discuss later.

It is also substantial that some early step in the compiler does not emit assembly too early, such as the issue with the gimplifier in GCC, or else the output file will be incomplete. We had to fix the gimplifier to avoid that.

Once the partition is applied to the child process and the output file was communicated to the driver, the compilation can continue. It should be running in parallel now by the number of partitions.

D. Name Clash Resolution

In LTO, if there are two promoted symbols to global with the same (mangled) name, a straightforward way to fix that is to increment a counter for each clashed symbol. This is possible because LTO has the entire program context, which we do not. Therefore, we need another way of fixing it.

Two naive approaches would be: (i) to select a random integer and append it to the function's name; (ii) to append the memory address of the function object to the function's name. Both break bootstrap [25]; the first is because output functions will have distinct names on every new compilation; the second is due to the fact that stage 1 compiles with `-O0` and stage 2 with `-O2`, which changes the memory address of the functions between these stages.

Our solution is to use a crc32 hash of the original file and append it to the function's name. There is still a tiny probability of name clash with this approach; however, we did not find any on our tests.

E. Integration with GNU Make Jobserver

GNU Make can launch jobs in parallel by using `make -j`. To avoid unnecessary partitioning and job creation when the CPU is at 100% usage, we have also implemented an integration mechanism with the GNU Jobserver [26]. The implementation is simple: we query the server for an extra token. If we receive the token, it means that some processor is available, and we can partition the TU and launch jobs inside the compiler. Else, the processor workload is full, and it may be better to avoid partitioning altogether.

F. Relationship with Reproducible Builds

One interesting point about Open Source projects is that they can be verified by everyone. However, frequently these projects are distributed in a binary form to the users, removing from them the burden of this process. Despite that, nothing avoids that a malicious developer modifies the codebase *before* the distribution (*e.g.* inserting a backdoor) and claiming that he/she got a distinct binary because his/her system is different from the user.

The *Reproducible Builds project* aims to solve that issue by providing a way to reproduce the released binary from its source. Some software needs to be patched to work with Reproducible Builds to not include temporal-dependent information, *e.g.* some kind of build timestamp. A build is called *reproducible* if, given the same source code and build instructions, anyone can recreate a bit-perfect version of the distributed binary [27].

We claim that our modification still supports the Reproducible Builds because of the following reasons:

- No random information is necessary to solve name clashing.
- Given a number of jobs, our partitioner will always generate the same number of partitions for a stated program, always with the same content.
- Partial Linking is always done in the same order. To ensure that, we communicate to the driver a pair (*partition number, path to file*), and we sort this list using the partition number as the key.
- No other race conditions are introduced, as we rely on the quality of the LTO implementation of the compiler.

However, there is one concern point, which is the Jobserver Integration. If the processor is already in 100% usage, we avoid partitioning absolutely and proceed with sequential compilation. This certainly changes accordingly to the processor usage of the machine during the build; therefore, the build is not guaranteed to be reproducible if this option is enabled. This is not an issue if the number of jobs is determined beforehand.

V. Correctness of the Method

We ensure the correctness of our changes by (1) bootstrapping GCC; (2) guaranteeing that the GCC test suite was passing; and (3) compiling random programs generated with *csmith* [28] (a tool that can generate random C programs that conform to the C99 standard) and ensuring that the correct result was found. These tests were done with 2, 4, 8, and 64 parallel jobs, with a minimum partitioning quota of 10^3 and 10^5 instructions.

For the time measurements, all points represent the average of collected samples, with the error bar representing a 95% confidence interval. Our test files consisted of preprocessed source files from GCC, which compiles without external includes. We collected $n = 15$ samples for every one of these files. Regarding the projects, we collected $n = 5$ samples because compilation is a computer-intensive task. We assured that in every test, `-g0` was passed to avoid duplication of debug symbols on nodes in other partitions.

Tests were executed on two computers, one with a Core-i7 8650U with 4 cores (8 threads), 8 GB of RAM, and an SSD storage drive, the other with 4x Opteron 6376 with 32 cores (64 threads), 256 GB of RAM, and an HDD storage drive. The code used in our tests is available at the official GCC git repository `git://gcc.gnu.org/git/gcc.git` with hash `e2da2f7205`.

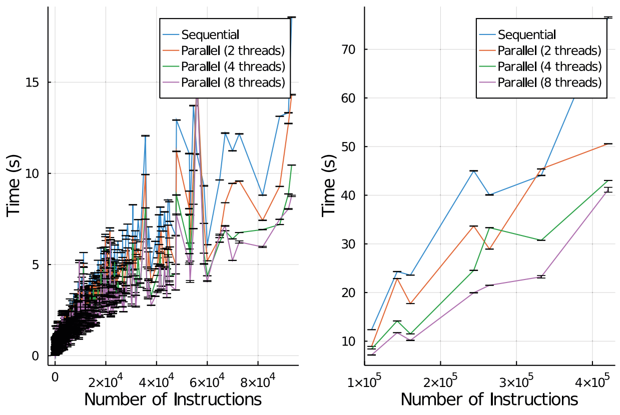

Fig. 5. Compilation of each file with 1, 2, 4, and 8 threads on Core-i7

VI. Results

We first highlight our best results in Table I. On this table, the autogenerated column means that this file is compiled to C++ and then compiled into assembly by GCC. We managed speedups of up to $2.4\times$ on Core-i7 when compiling individual files with 8 threads and up to $3.53\times$ on Opteron 6376 when with 64 threads. Figure 5 shows the results of all files in GCC. Here it is possible to observe that for files with Number of Instructions $> 1 \times 10^5$, we have the most significant improvements.

From now on, we will discuss how our proposed changes impact the overall compilation time of some projects. For this, we conducted experiments compiling the Linux Kernel 5.19.6, Git 2.30.0, the GCC version mentioned in Section V with and without bootstrap enabled and JSON C++, with commit hash `01f6b2e7418537`. We have only enabled Jobserver integration in GCC and Git because it is necessary to modify an absolute large number of Makefiles to do so (for instance, Linux has 2597 Makefiles).

Figure 6 shows our results. We can observe a near 35% improvement when compiling GCC with bootstrap disabled, 25% when bootstrap is enabled, and 15% improvement when compiling Git compared to `make -j64` alone. Our Jobserver implementation offers a small improvement in GCC compilation but showed a massive slowdown in Git. This is due to the fact that Jobserver integration has an interprocess communication cost with Make, which is a problem if the size of the partitions is small. These tests were executed with 64 Makefile jobs and 8 threads inside the compiler.

VII. Conclusions and Future Works

We have shown a tangible way of compiling files in parallel, capable of being implemented in an industrial compiler, which resulted in speedups when compiling single files in parallel, as well as some speedups when compiling entire projects in many-core machines.

	Core-i7			Opteron 6376			
File	Sequential	8 Threads	Speedup	Sequential	64 Threads	Speedup	Autogenerated
gimple-match.c	$76s$	$31s$	$2.4\times$	$221s$	$66s$	$3.32\times$	yes
insn-emit.c	$23s$	$10s$	$2.25\times$	$97s$	$37s$	$3.53\times$	yes
tree-vect-stmt.c	$11s$	$5s$	$2.14\times$	$32s$	$13s$	$2.46\times$	no

TABLE I

SPEEDUP OF HIGHLIGHTED FILES

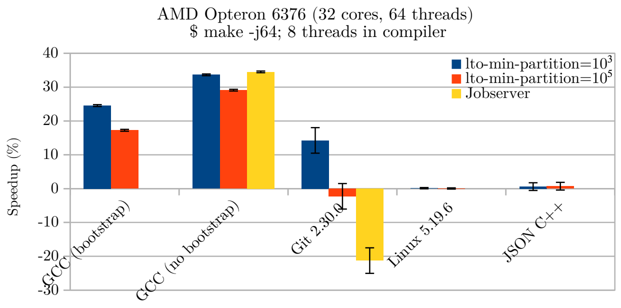

Fig. 6. Compilation of some projects with 64 Makefile jobs and 8 threads in compiler in Opteron 6376

This project has several possibilities for future work. We intend to improve the load balancing algorithm of the partitioner using LTO's current partitioner algorithm as its base. We also plan to modify the driver to support external compilers through the GCC SPEC language. Our current implementation only checks if the launching program is a known compiler/assembler/linker and will not work for languages that need additional steps, such as CUDA.

Another interesting work would be to develop a predictive model to decide if the input file is a good candidate for parallel compilation. Figure 5 shows a clear linear correlation between the expected number of instructions and time (and maybe it is the best parameter). Still, it may be possible to (statically) collect more information about the file for a better decision.

Finally, we will work on the parallelization of the LTO's WPA step, which currently runs sequentially. Profiling shows that 11% of the compilation time is spent in this step, while our project parallelized 75% of the compiler's work.

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2nd ed. Addison-Wesley, 2007.

[2] GNU Collaborators, "GCC releases history," 2019. [Online]. Available: https://www.gnu.org/software/gcc/releases.html

[3] ——, "Link-time optimization in GCC: Requirements and high-level design," 2005. [Online]. Available: https://gcc.gnu.org/projects/lto/lto.pdf

[4] P. Briggs, D. Evans, B. Grant, R. Hundt, W. Maddox, D. Novillo, S. Park, D. Sehr, I. Taylor, and O. Wild, "WHOPR – fast and scalable whole program optimizations in GCC," 2007. [Online]. Available: https://gcc.gnu.org/projects/lto/whopr.pdf

[5] T. Glek and J. Hubička, "Optimizing real world applications with GCC link time optimization," 2010. [Online]. Available: https://arxiv.org/abs/1010.2196

[6] GNU Collaborators, "GENERIC documentation," 2019, accessed: Jul. 5, 2022. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html

[7] G. Belinassi, "PR86829 missing sin(atan(x)) optimization," 2018, accessed: Jul. 5, 2022. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86829

[8] GNU Collaborators, "GCC GIMPLE documentation," 2020. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

[9] ——, "GCC RTL representation," 2019. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/RTL.html

[10] ——, "Improve instruction selection," 2008. [Online]. Available: https://gcc.gnu.org/wiki/Improve_instruction_selection

[11] N. Lincoln, "Parallel programming techniques for compilers," *SIGPLAN Not.*, vol. 5, no. 10, pp. 18–31, Oct. 1970.

[12] C. N. Fischer, "On parsing context free languages in parallel environments," Cornell University, Tech. Rep. TR 75-237, 1975. [Online]. Available: https://hdl.handle.net/1813/7121

[13] S. Fowler and J. Paul, "Parallel parsing: The earley and packrat algorithms," May 2009, accessed: Jun. 7, 2022. [Online]. Available: http://people.eecs.berkeley.edu/~kubitron/courses/cs252-S09/projects/reports/project5_report_ver2.pdf

[14] P. Dubroy and A. Warth, "Incremental packrat parsing," in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, 2017, pp. 14–25.

[15] A. Barenghi, S. C. Reghizzi, D. Mandrioli, F. Panella, and M. Pradella, "Parallel parsing made practical," *Science of Computer Programming*, vol. 112, pp. 195–226, Nov. 2015.

[16] M. T. Vandevoorde, "Parallel compilation on a tightly coupled multiprocessor," DEC Systems Research Center, Tech. Rep. SRC-RR-26, Mar. 1988.

[17] D. B. Wortman and M. D. Junkin, "A concurrent compiler for modula-2+," *ACM SIGPLAN Notices*, vol. 27, no. 7, pp. 68–81, Jul. 1992.

[18] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: A compiler infrastructure for the end of moore's law," 2020. [Online]. Available: https://arxiv.org/abs/2002.11054

[19] M. T. Bernardino, G. Belinassi, P. Meirelles, E. M. Guerra, and A. Goldman, "Improving parallelism in git and gcc: Strategies, difficulties, and lessons learned," *IEEE Software*, 2020.

[20] M. Liška, "Optimizing large applications," 2014. [Online]. Available: https://arxiv.org/abs/1403.6997

[21] D. X. Li, R. Ashok, and R. Hundt, "Lightweight feedback-directed cross-module optimization," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010, pp. 53–61.

[22] T. Johnson, M. Amini, and X. D. Li, "Thinlto: scalable and incremental lto," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 111–121.

[23] "Itanium C++ ABI," 2012, accessed: Jul. 5, 2022. [Online]. Available: https://itanium-cxx-abi.github.io/cxx-abi/abi.html#vague

[24] Z. Galil and G. F. Italiano, "Data structures and algorithms for disjoint set union problems," *ACM Computing Surveys*, vol. 23, no. 3, pp. 319–344, Sep. 1991.

[25] GNU Collaborators, "Installing GCC: building," 2021, accessed: Jul. 5, 2022. [Online]. Available: https://gcc.gnu.org/install/build.html

[26] ——, "Posix jobserver interaction," 2020, accessed: Jul. 5, 2022. [Online]. Available: https://www.gnu.org/software/make/manual/html_node/POSIX-Jobserver.html

[27] C. Lamb, V. R. Young, and C. Lang, "When is a build reproducible?" 2016, accessed: Jul. 5, 2022. [Online]. Available: https://reproducible-builds.org/docs/definition/

[28] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.