Compiling Files in Parallel: A Study with GCC*

Giuliano Belinassi[1], Richard Biener[2], Jan Hubička[2,3], and Alfredo Goldman[1]

[1] Institute of Mathematics and Statistics, Rua do Matão 1010, São Paulo SP, BRA
https://www.ime.usp.br
[2] SuSE Labs, Nürnberg 90409, GER
https://www.suse.com/
[3] Charles University, Malostransk én ám. 25. 11800 Praha, CZE
https://cuni.cz/uken-1.html

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× speedup when compiling individual files. We also found no meaningful slowdown when compiling other applications with Jobserver disabled. We also explain why the adoption of our proposed method is still compatible with the Reproducible Builds project.

Keywords: Compilers · Parallel Compilation · Link Time Optimization · LTO.

1 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 the development of them, it is true that their code will be, at some point, transformed into machine language by a compiler or assembler, even if an interpreter executes it.

Compilers are enormous programs, largely adopted by industry and academia, where a great effort has been employed to produce efficient code – but without any sacrifice in correctness –. There are huge projects destined to develop and improve them, such as the GNU Compiler Collections (GCC[4]) and LLVM[5], capable of translating several languages such as C, C++, and Fortran, to machine language.

* Supported by CAPES and Google Summer of Code.
[4] https://gcc.gnu.org/
[5] https://llvm.org/

GCC was started by Richard Stallman, with the first public release in March of 1987. Back then, it only supported the C language, but already allowed code generation for several architectures [8]. Today, GCC is a multinational collaboration project, with more than 412230 lines of code, and perhaps the most used C/C++ compiler in Linux environments.

GCC was Initially designed to compile programs a single file at time, meaning that it could not allow global cross-file optimizations because the compiler never had the opportunity to analyze the program as a whole. This scenario changed when Link Time Optimization (LTO) was proposed [4] and implemented in GCC [7]. GCC supports LTO by using `-flto`. Part of the LTO engine has already been parallelized, which inspired this work.

This work tries to answer the question about how can we modify a industrial scale compiler to compile single files in parallel. We address that by reusing the already-existing LTO engine in it (in this case, GCC) for partitioning the single file Compilation Unit after the Interprocedural Optimizations has been decided, and we proceed compiling each partition individually, in parallel.

We present the previous efforts in compiling a single file in parallel, as well as an introduction to LTO in Section 2. Then we detail our work in single file compilation in Section 3, while exposing some internal mechanisms of GCC. We then show how we ensured that our modifications are correct in Section 4. Finally, we present our results in Section 5 and discuss how to improve from this paper in Section 6.

2 Related Works

Parallel Compilation includes parsing (parallel or not), how to perform analysis, optimization, and code translation in parallel. Given an alphabet Σ, parsing can be described as building a machine to decide if an input string $w \in \Sigma^*$ is a member of a certain language $L \subseteq \Sigma^*$ or not, creating the Abstract Syntax Tree in the process by logging the used derivation rules.

Parallel Parsing dates back to 1970. Lincoln [10] explored how to use the vectorial registers in the (so far) STAR-100 supercomputer for lexical analysis. Fischer [6] gives a detailed theoretical study, proving several concurrent parsing techniques for LR(k), LALR(k) and SLR(k). The parser proceeds by breaking the input in 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 5 possible cases. However, in case of an error, the parser result should be discarded, and therefore a lot of work will be done in vain when comparing with the sequential version.

Perhaps the most interesting work is by Barenghi *et. al.* [2], where they explore properties of Operator Precedence Grammars to construct an 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× when compared to a parser generated by GNU Bison.

As for parallel compilation *de facto*, we found two relevant works by Vandevoorde [12] for C, and Wortman and Junkin [13] for Modula-2+. The former assumes that every function declaration is in the file headers, and implements per-function and per-statement parallel compilation. The letter implements only per-function parallelism. Speedups ranged from 1.5× to 6× on a multicore MicroVAX II machine. None of these papers discuss optimization, and they concentrate on (today perspective) non-optimizer compilers, which are not the case of GCC.

There has been an attempt of parallelizing GCC by threading the GIMPLE Intraprocedural pass manager [3]. An optimization is called Intraprocedural if it only requires information contained inside its body. The *vectorizer* is an example of this. The authors managed a speedup of up to 3.35× to this compilation stage, and up to 1.88× in total compilation of a file when extending this technique to the RTL passes.

2.1 Link Time Optimization (LTO)

Compilation usually uses the following scheme: a compiler consumes a source file, generating an assembler 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. Fig. 1 illustrates this process for a single file. In this paper, we call this method the Classical Compilation scheme.

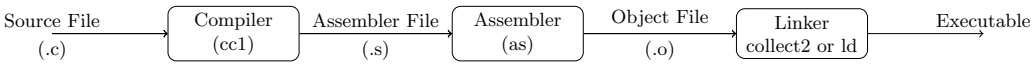

Fig. 1. GCC Compiling a .c file in Classical Compilation mode

The issue around the Classical Compilation scheme is that it can only optimize with the information found in its Compilation Unit because it can not see the body content of functions in other files of the project. A Compilation Unit is the entire content of a source file (a .c file in C) plus all headers it includes.

As an answer to this, LTO allows cross-module optimizations by postponing optimizations and final translation to a linker wrapper. There, the entire program can be loaded by the compiler (but more often, just some sort of summary) as a single, big Compilation Unit, and optimizations can be decided globally, as now it has access to the internals of other modules. LTO is divided into three steps [4,7]:

- LGEN (*Local Generation*): each module is translated to an Intermediate Representation (IR) and written to disk in phony object files. These objects do not contain assembler code, and therefore can not be linked by a default linker such as *ld* (unless fat objects are enabled, in this case, the assembler is also generated and dumped together with the IR). This step runs serially on the input file (*i.e.* in parallel with regard to the files in the project).

– WPA (*Whole Program Analysis*): load all translated modules, merges all Compilation Units into one, and analyzes the program globally. Then it generates a log of transformations for the program, and this global Compilation Unit is partitioned 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 will generate its own object file. This stage runs in parallel.

This process is sketched in Fig. 2, where the linker wrapper is represented by *collect2*, which firstly launch *lto1* in WPA mode, and the second time it finally launches *ld*. This process can be seen by launching gcc with `-flto -v`.

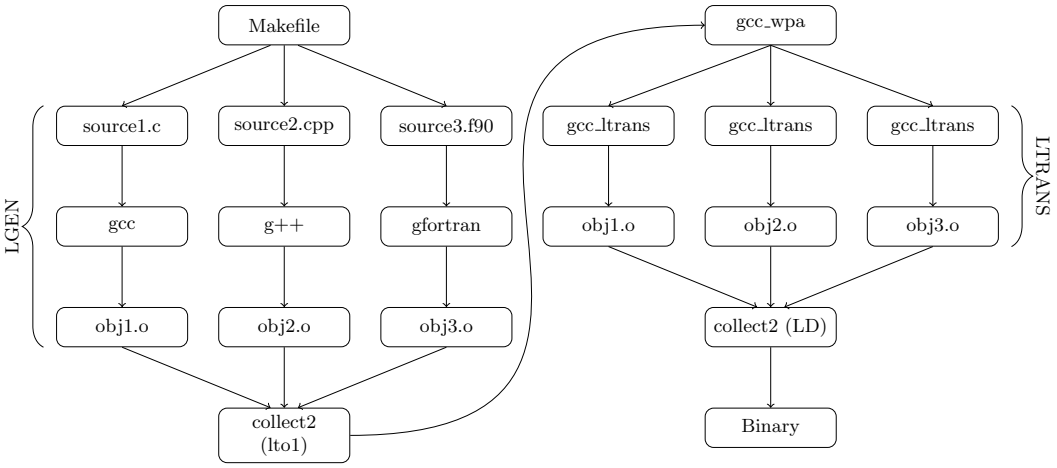

Fig. 2. Compilation of a program using LTO scheme

3 Our Proposal

As presented in Section 2.1, LTO was created to allow cross-module optimizations in programs, and has a serial part (WPA), imposing a bottleneck on many-core machines. On the other hand, Classical Compilation scheme can not partition its Compilation Unit for parallel compilation, which can also bottleneck the compilation 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 claim that it is possible, and show that by implementing it in GCC.

Our approach differs from LTO mainly in how we handle the Interprocedural Optimization. An optimization is called Interprocedural if it requires information of the body of other functions to optimize a certain function. An example of

such optimizations is the *inliner*. LTO handles these optimizations with the context of the whole program, while our approach will only have the context of the original Compilation Unit. This allows optimizations as good as they are in the Classical Compilation scheme while benefiting of the extra parallelism opportunity available in the LTO's LTRANS stage.

In this section, we will first discuss the internals of some parts of GCC, which we had to modify for our implementation to work. In Subsection 3.1, we present an important piece of the compiler from the User Experience perspective: the `gcc` *driver*. Then, in Subsection 3.2 we present a short algorithm for making the LTO partitioner work for our proposal. In Subsection 3.3 we present a necessary change we had to do in our work about how partitions are applied in GCC. In Subsection 3.4, we explain how we solved the issue of private symbols with the same name being promoted to global. Then in Subsection 3.5 we present an opitional part of our work about communicating with the GNU Make Jobserver to keep track of used processors. And finally, we discuss why our proposal is still compatible with the Reproducible Builds project in Subsection 3.6.

3.1 The GCC driver

A large program can be written in several languages, with each of them having its own compiler. From a Compiler Theory perspective, a compiler translates a program from a language A to an language B [1]. In GCC, it translates several languages, such as C, to Assembler of some architecture (*e.g.* x86). This means that encapsulating code in object files, or linking the code in an executable, are not tasks of the compiler. However, the user can launch `gcc -o binary file.c` and get a working binary. That is because the binary `gcc` is a *driver*, and it will launch the necessary programs for the user. In fact this line launches three programs, as illustrated in Fig. 1.

Therefore, if we want our changes do not break the building scripts (*e.g.*, Makefile) used by most projects – which is mostly launching `gcc file.c -c` and creating an object file `file.o` – we must ensure that we create a single object file for each file, not multiple, as does the LTO partitioner. Fortunately we can rely on GNU *ld* partial linking for merging objects file into one. Therefore, the solution to this problem is:

1. Patch the *partitioner* to communicate the location of each file created to the *driver*. If the *partitioner* is the compiler (which is the case of GCC), then it should communicate the location of each generated *assembler file*. This can be archived by passing a hidden flag `-fadditional-asm=<file>` by the driver to the partitioner, which the last will write to. This file can also be replaced with a Named Pipe for better performance if needed.
   Then, the partitioner checks if this flag has been passed to the compiler. If yes, then a *compatible version* of the driver is installed. If the partitioner decides to partition the Compilation Unit, it should *retarget* the destination assembler file and write the retargeted name to the communication file.

2. Patch the driver to pass this hidden flag to the *partitioner*, and also check if this file exists. If not, it means that either the compiler is incompatible (assuming it did not halt with an error) or it has chosen not to partition the Compilation Unit. In the first case, the driver should call *as* to every assembler file generated, and call the linker to generate the expected final object file. In the second case, simply fallback to the previous compilation logic.

Fig. 3 illustrates the code flow after these changes. The execution starts in the highlighted node *Driver*, which calls the compiler with the necessary flag to establish a communication between the parts. The compiler then will partition the Compilation Unit and forks itself into several child processes, one for each partition. Although there is a clear difference between a *process* and a *thread*, we will use this term interchangeable in this work.

Once multiple processes are created, the the compiler will communicate its output .s file to the driver, and the driver then will launch the *as* to assemble it, and then launch *ld* to merge them all into a single object file.

After these changes, a good way to check if the changes are working is to bootstrap the compiler with a single partition, but writing the output path into the created communication channel. Bootstrapping can be a resource intensive task; therefore this is an excellent opportunity to write automated tests covering every case necessary for the bootstrap. This may also expose some extreme cases, for instance, the C compiler being called to process macros in files of distinct languages.

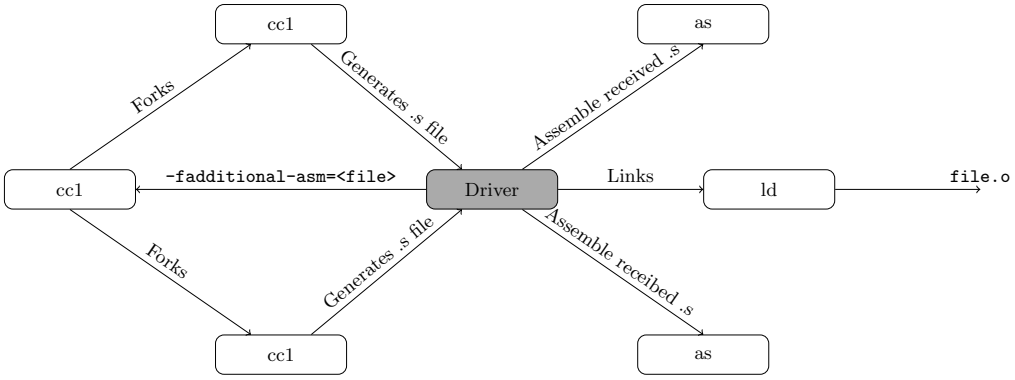

Fig. 3. Interaction between the *driver*, the *compiler*, and other tools after our changes

3.2 Adapting the LTO Partitioner

In GCC, a Compilation Unit is represented as a callgraph. Every function, global variable, or clones (which may represent a function to be inlined) is represented

as nodes in a callgraph. If there is a function call from f to g, then there is an edge from f to g. Similarly, if there is a reference to an global variable v in f, then there is also an edge from f to v. This means that Compilation Unit 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 it (*e.g.* the size in lines of code it had). This is done to conserve memory. However, when LTO is disabled, the function body is present, resulting in some assertion failures, which were fixed after some debugging.

Then comes the partitioner algorithm *de facto*. In LTO, GCC tries to create partitions of similar size, and always try to keep nodes together. The heuristic used by the original partitioner runs in linear time, and because of that its code is quite complex. This resulted in a difficult process of finding and solving the issues causing a partitioning failure, and therefore, we decided to design a new partitioning algorithm for this project.

This partitioning algorithm works as follows: for each node, we check if they are a member of a COMDAT group, and merged it into the same partition. We then propagate outside of this COMDAT group; checking for every node that may trigger the COMDAT to be copied into other partitions, and also add them to 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 of the group. Fig. 4 represents a sketch of this process.

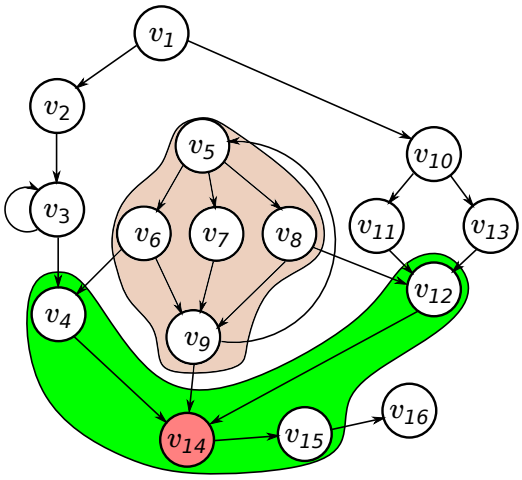

Fig. 4. Example of callgraph, in beige being represented the COMDAT group, in green the COMDAT frontier, and in red the cloned nodes.

At first we also did this process for private functions 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 hit in any parallelism opportunity. For grouping the nodes together, we used an Union Find with Path Compression, 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 callgraph [5].

Once the partitions are computed, we need to compute its *boundary*. If function f calls g, but they were assigned into distinct partitions, then 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, some 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. Second being variables marked as in the boundary (and therefore not in the partition) not being correctly promoted to external. The reason for these issues were hard to find the reason of, but easy to fix.

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

3.3 Applying a Partition Mask

Once partitions are computed, the only presented way to apply it (*i.e.*, remove every unnecessary node from the Compilation Unit) was to reload the compiler, and let it load the phony object files, which are not available 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 keep its body*: we mark that this function is available in other partition, but we do not release its body or datastructures.
- *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 it callees, all references to variables, the content of the Dominator Tree, and also its Control-Flow Graph. This is now a function which this partition only know that it exists.
- *Node not in boundary*: we remove the node, and all its content.

After this, it retargets the output assembler file to another file private to this partition and write its partition number together with the path to the communication file, which the driver will read in the future. The partition number is important to guarantee that the build is reproducible, as we will discuss later.

It is also important that some early step in the compiler does not emit assembler 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 current process, and the output file has been communicated to the driver, it can continue with the compilation. It should be running in parallel now by the number of partitions.

3.4 Name Clash Resolution

In LTO, if there are two promoted symbols to global with the same name, it is quite straightforward to fix this issue. Since we have the context of the entire program, we can simply increment a counter for each clashed symbol and rename them. However, we have not the context of the entire program, and we have to find another way of fixing it.

One will be tempted to simply select an random integer and append to the name. This is not a good idea, once it will break bootstrap because each compilation will result in a different object file. Using the address of the function in memory does not work either, as the first bootstrap step has the compiler compiled with `-O0` and the second step with `-O2`, and memory address will certainly be different.

The solution is to use a crc32 hash of the original file and append it to the function's name, since we can not have two functions with the same (mangled) name in the program. There is still a tiny probability of name clash with this approach, however we did not find any on our tests.

3.5 Integration with GNU Make Jobserver

GNU Make is able to launch jobs in parallel by using `make -j`. This is so simple and yet so powerful that even recent building scripts (such as CMake) relies on Make for launching jobs.

In order to avoid unnecessary partitioning and job creation, we have also implemented a integration mechanism with GNU Jobserver [11]. The implementation is simple: we query the server for an extra token. If we receive the token, then it means that some processor is available, and we can partition the Compilation Unit and launch jobs inside the compiler. Else, the processor workload is full, and it may be better to avoid partitioning altogether.

However, for a program to be able to communicate with the jobserver, it should be launched with a prepended `+` character, for example `+gcc -c file.c`, and therefore it is not so straightforward to use this mode on existing projects.

3.6 Relationship with Reproducible Builds

One interesting point of Open Source is that it can be verified by everyone. However, very often these projects are distributed in a binary form to the users, removing them the burden of this process. But there is nothing avoiding that a

malicious developer modifies the codebase *before* the distribution (*e.g.* inserting a backdoor), and claiming that he got a distinct binary because his/her system is different from the user.

The Reproducible Builds aims to solve that issue by providing a way to reproduce the released binary from its source. Some software needs to be patched in order to work with Reproducible Builds, for instance, to not contain some kind of build timestamp, and so on. A build is called *reproducible* if given the same sourcecode and build instructions, anyone can recreate a bit-perfect version of the distributed binary [9].

To keep compatibility with this project, we must ensure that our compiler always output the same code with a given input. The input is not only the source file itself, but also the flags passed to it.

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

1. No random information is necessary to solve name clashing.
2. Given a number of jobs, our partitioner will always generate the same number of partitions for a certain program, always with the same content.
3. 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 key.
4. No other race conditions are introduced, as we rely on the quality of the LTO implementation of the compiler.

However, there is one point of concern, which is the Jobserver Integration. If the processor is already in 100% usage, we avoid partitioning at all and proceed with sequential compilation. This certainly changes from computer to computer, and 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.

4 Methods and Current Issues

We ensure the correctness of our changes by:

1. Bootstrap GCC with the number of parallel jobs as 2, 4, 8 and 64, with minimum partitioning quota of 10^3 and 10^5 instructions. We found issues with regard to how the *ipa-split* pass generated clones, and therefore we have disabled it for now. This pass breaks big functions into a header and a body, and marks the header to be inlined to the caller. A bug in our implementation marked such clones as being an external function in rare cases, which the linker can not find.
2. Run the GCC testsuite, which we noticed that all tests related to the debug symbols were failing. However, if `-g0` is passed, no debug symbol is created, and the bug related to this issue is never triggered. The reason behind this is because our partition applier do not remove the symbols associated with removed nodes, resulting in unknown symbol being dumped into assembler.

3. Generated random programs with *csmith* [14]. This found more complicated bugs (for instance, long strings being output before the assembler file retarget), which we fixed.

Then for the time measurements, we sampled $n = 15$ points for each file from the GCC project, which we present the average in the graphics. The errorbars represents 95% confidence interval for the sample average. Furthermore, each file were preprocessed by the C preprocessor, so it could be compiled independently of other files.

For the projects, each point represents a mean were compiled with $n = 5$ points, because this is a more computer intensive task. The errorbars also represent a 95% confidence interval of the average. We also were sure that the serial data was also compiled with `-g0` for fairness.

Tests were mainly executed in two computers, which are represented in the Table 1. The graphic caption specify where the test was run.

The version of GCC used in the tests is available in the `autopar_europar_2021` branch of `git://gcc.gnu.org/git/gcc.git`, with hash `e2da2f7205`.

	Number of Cores	Number of Threads	RAM	Disk Type
Core-i7 8650U	4	8	8Gb DDR4	SSD
4x Opteron 6376	32	64	128Gb DDR3	HDD

Table 1. Machine specification of tests

5 Results

We first highlight our best results. We managed a 2.4× and 2.25× speedup on the Core-i7 machine when compiling the files *gimple-match.c* and *insn-emit.c*, which are autogenerated files from the GCC project, and takes quite long to compile (76*s* and 23*s* sequential). For non-generated files, we have *tree-vect-stmt.c*, with a 2.14× speedup and taking 11*s* sequentially. All these speedups were archived by using 8 parallel jobs. Fig. 5 shows the results globally. Here we can see that for files with *Number of Instructions* $> 1 \times 10^5$, we have mostly significant improvements.

Now we move to the 4× Opteron machine. We managed a 3.32×, 3.53× on *gimple-match.c* and *insn-emit.c* by using 64 jobs. This is close to the maximum theoretical speedup of 4× computed by [3] when parallelizing the Intraprocedural Optimizations.

We will now discuss how these changes impacts the overall compilation time of some projects. For this, we run experiments compiling the Linux Kernel 5.19.6, Git 2.30.0, the GCC version mentioned in Section 4 with and without bootstrap

enabled, and JSON C++, with commit hash `01f6b2e7418537`. We have also compiled GCC and Git with the jobserver integration enabled, and the reason because we did not enabled it in other projects is because it is necessary to modify a absolutely large number of Makefiles (for instance, Linux has 2597 Makefiles).

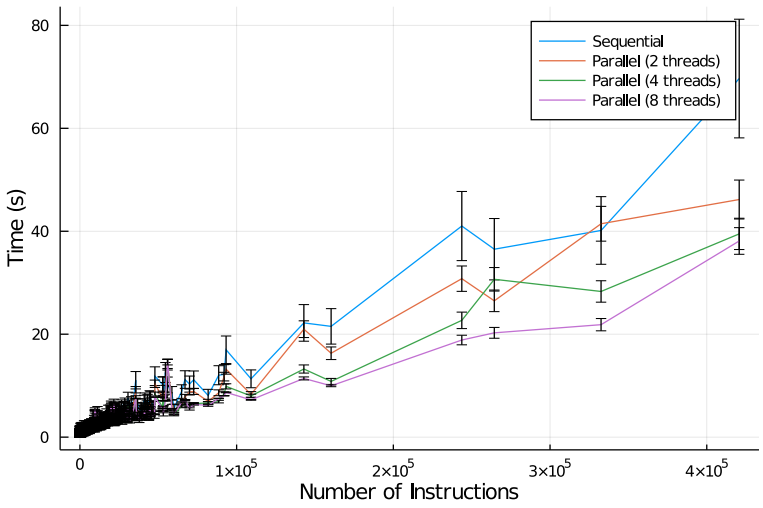

Fig. 5. Compilation of each file with 1, 2, 4, and 8 threads on Core-i7

In Fig. 6 we show our results. We can observe a near 35% improvement when compiling GCC with bootstrap disabled, a 25% when bootstrap is enabled, and 15% improvement when compiling Git with respect to `make -j64` alone. Our jobserver implementation also squeezed a small improvement in GCC compilation, but yielded a massive slowdown in Git. This is because Jobserver integration comes with a cost of IPC with Make, which is a problem if the size of the partitions are small. These tests were executed with 64 Makefile jobs and 8 threads inside the compiler. Other than this, we seen no significant speedup or slowdown in these other projects.

6 Conclusions and Future Works

We have shown a tangible way of compiling files in parallel capable of being implemented in industrial compilers, which resulted in speedups when compiling single files in parallel, as well as some speedup when compiling entire projects in manycore machines. However, There are several points in which our work can be improved.

First, is by fixing the problems reported in Section 4. These bugs certainly prevents the current branch to be used in industrial environments (which is a

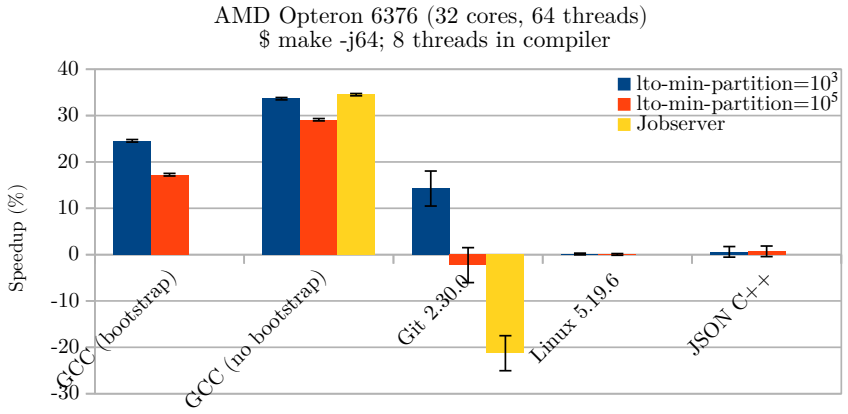

Fig. 6. Compilation of some projects with 64 Makefile jobs and 8 threads in compiler

good reason why this was not merged in upstream yet), but they are fine as a proof of concept to support our claims.

Second is by implementing a better partitioner to the project. One main issue with our partitioner is that we kept its load balancing algorithm minimal to ensure that it works. Using the LTO default partitioner as a base is a good start.

Third, by modifying the driver to also support external compiler through GCC SPEC language. Current implementation only checks if launching program is a known compier/assembler/linker, and will get confused in languages that needs additional steps (such as CUDA).

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

We also would like to draw attention to LTO's WPA step, which still runs sequentially. Parallelizing this step, which includes all interprocedural optimization passes would improve parallel compilation of projects across the board in both LTO mode and in our work. Profiling shows us that 11% of the compilation time is spent in this mode, while our project parallelized 75% of the compiler.

References

1. Alfred V. Aho, Ravi Sethi, and D. Jeffrey Ullman. Compilers, principles, techniques, and tools. second edition, 2007.
2. Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. Parallel parsing made practical. *Sci. Comput. Program.*, 112(P3):195–226, November 2015.

3. Matheus Tavares Bernardino, Giuliano Belinassi, Paulo Meirelles, Eduardo Martins Guerra, and Alfredo Goldman. Improving parallelism in git and gcc: Strategies, difficulties, and lessons learned. *IEEE Software*, 2020.
4. Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, and Ollie Wild. Whopr - fast and scalable whole program optimizations in gcc, 2007. URL: https://gcc.gnu.org/projects/lto/whopr.pdf.
5. Paulo Feufiloff. Union-find, 2002. URL: https://www.ime.usp.br/~pf/mac0122-2002/aulas/union-find.html.
6. Charles N Fischer. On parsing context free languages in parallel environments. Technical report, Cornell University, 1975.
7. T. Glek and Jan Hubička. Optimizing real world applications with gcc link time optimization. *ArXiv*, abs/1010.2196, 2010.
8. GNU. Gcc releases history, 2019. URL: https://www.gnu.org/software/gcc/releases.html.
9. Chris Lamb, Valerie R. Young, and Clemens Lang. When is a build reproducible?, 2016. URL: https://reproducible-builds.org/docs/definition/.
10. Neil Lincoln. Parallel programming techniques for compilers. *SIGPLAN Not.*, 5(10):18–31, October 1970.
11. GNU Organization. Posix jobserver interaction, 2020. URL: https://www.gnu.org/software/make/manual/html_node/POSIX-Jobserver.html.
12. Mark Thierry Vandevoorde. *Parallel compilation on a tightly coupled multiprocessor*. Systems Research Center, 1988.
13. David B. Wortman and Michael D. Junkin. A concurrent compiler for modula-2+. *SIGPLAN Not.*, 27(7):68–81, 1992.
14. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.