Compiling Files in Parallel: A Study with GCC

Anonymous Author(s)

ABSTRACT

Processors are becoming increasingly parallel, but compiling software has so far been a task parallelizable only by per file granularity. To improve this, we propose a method feasible to implement in commercial compilers for parallel single file compilation, using simple modifications in the Link Time Optimization (LTO) engine; which we show by implementing a custom partitioner in GCC. This method resulted in a 35% speedup when self-compiling GCC when compared to `make -j` only parallelism, and speedups ranging from 0.7× to 3.5× when compiling individual files. We also explain why the adoption of our proposed method is still compatible with the Reproducible Builds project.

CCS CONCEPTS

• **Software and its Engineering** → **Compilers**; • **Computer methodologies** → *Parallel algorithms.*

KEYWORDS

Compilers, Parallel Compilation, Link Time Optimization, LTO, GCC

ACM Reference Format:

Anonymous Author(s). 2018. Compiling Files in Parallel: A Study with GCC. In *ICPP '21: 50th International Conference on Parallel Processing, August 09–12, 2021.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

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 by coding new websites in scripting languages, creating standalone desktop applications, adding new functionalities to an operating system, and so on. 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)[1] and LLVM[2], capable of translating several languages such as C, C++, and Fortran, to machine language.

GCC was started by Richard Stallman, with the first public release in March of 1987, supporting only the C language and targeting several architectures [15]. Today, GCC is a multinational 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 there was not any data structure that allowed analyzing the program as a whole. This scenario changed when Link Time Optimization (LTO) was proposed [5, 6] and implemented [14]. GCC supports LTO by using `-flto`. Part of the LTO engine has already been parallelized, which inspired this work.

The main motivation of this work is to speed up the compilation of softwares that uses blob files, which might severely impact the compilation opportunity when using `make -j` only parallelism, imposing a bottleneck. This is very noticeable on high-core count machines, as illustrated by Figure 1. In this figure, we collect the compilation timestamp on the beginning and end of each GCC file, which resulted in this interval graph. One blob file responsible for such bottleneck in `gimple-match.c`, a file that holds several search-and-replace patterns for tree optimizations. We can also conclude that GCC has a compilation bottleneck due to these blob files.

The main contribution of this work is to answer the question about how can we modify an 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 Translation Unit (TU) after the Interprocedural Analysis (IPA) has been decided, and we proceed compiling each partition individually, in parallel. IPA includes only interprocedural optimizations, which requires 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 the flags `-flto -flinker-output=nolto-rel`, but with phony object creation overhead and no guarantee of correct name clash resolution, as discussed in Section 4.4.

We present the previous efforts in compiling a single file in parallel, as well as an introduction to LTO in Section 2. Then, discuss the idea that motivated the development of this feature on Section 3, reproducing previous profiling experiments on GCC and computing an approximation about the best speedup archivable by our methods; to finally present our proposal on Section 4, as well as presenting some internal details of GCC. We then show how we ensured that our modifications are correct in Section 5; and finally, we present our results in Section 6 with a discussion about our findings and observations when running the experiments. In the end, we discuss how to improve from this paper in Section 7.

ICPP '21, August 09–12, 2021, Chicago, IL

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

[1] https://gcc.gnu.org/
[2] https://llvm.org/

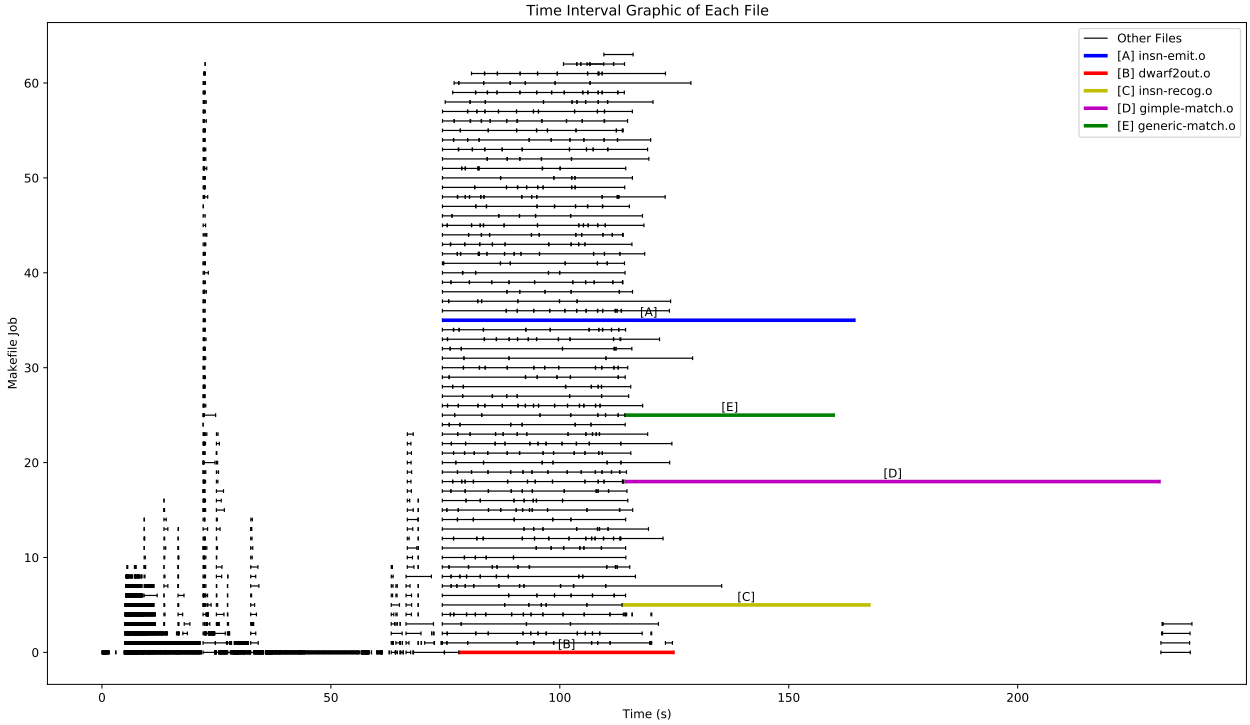

Figure 1: Compilation time of GCC without our modifications

2 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 certain language or not, creating the Abstract Syntax Tree in the process by logging the used derivation rules.

Parallel Parsing dates back to 1970. Lincoln [20] 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 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.

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. For solving 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 thread on its right. The authors managed a speedup of 5.5× in Earley, and 2.5× in Packrat.

Still on Packrat methods, Dubroy and Warth [10] show how to implement an incremental Pacrkat to avoids reparsing the entire input on modifications. The authors achieve this by recording all its intermediate results in a parsing table and modify the parsing program to reload this table when invoked. They further showed that their method does not require any modification to the original grammar, and their method requires only up to 11.7% of extra memory when compared to the original parser. Authors claim a reduction from 23.7ms to 6.2ms on reparsing a 279Kb input string.

Barenghi *et al.* [3] 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× when compared to a parser generated by GNU Bison.

As for parallel compilation *de facto*, works dates back from 1988. Vandevoorde [22] worked on a C compiler, and Wortman and Junkin [23] 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× to 6× on a multicore MicroVAX II machine. None of these papers discuss optimization, and they concentrate on (today's perspective) non-optimizer compilers, which are not the case of GCC.

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

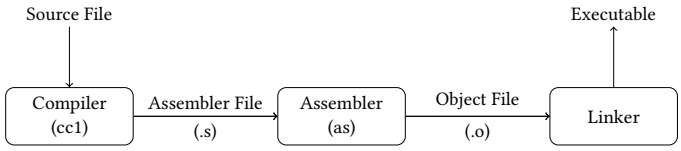

Figure 2: GCC Compiling a .c file in *classical compilation* mode

Panchenko *et al.* [21] presented Lightning BOLT, a compiler that reads already compiled binaries and outputs an optimized version of them. The authors present a list of passes in which Lightning BOLT runs in parallel and briefly explain how they parallelized them. They include Control-Flow Graph Construction, Local Optimizations, Identical Code Folding, Frame optimizations, and Shrink Wrapping running in parallel. Speedups ranged up to 1.56× when compared to the sequential version.

There has been an attempt of parallelizing GCC by threading the GIMPLE intraprocedural pass manager [4], which only requires information contained inside the function's body (*e.g.* vectorization). 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 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 2 illustrates this process for a single file. In this paper, we call this method the *classical compilation* scheme.

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

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 TU, and optimizations can be decided globally, as now it has access to the internals of other modules. LTO is divided into three steps [5, 14]:

- LGEN (*Local Generation*): each module is translated to an IR 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*): load all translated modules, merges all TUs into one, and analyzes the program globally. After that, it generates an *optimization summary* for the program and partitioned 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 will generate its own object file. This stage runs in parallel.

This process is sketched in Figure 3, where the linker wrapper is represented by *collect2*, which firstly launch *lto1* in WPA mode, and

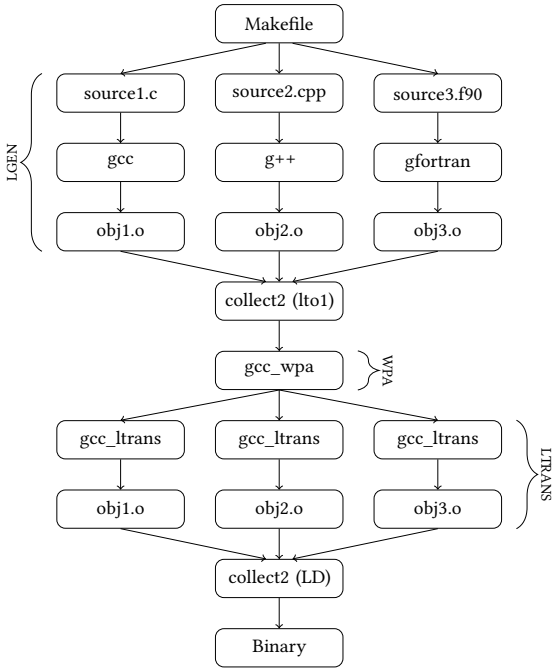

Figure 3: Compilation of a program using LTO scheme

then it finally launches *ld*. This process can be seen by launching gcc with `-flto -v`.

There are also papers using a profile-based approach rather than a whole-program analysis for cross-module optimizations. Xinliang *et al.* [19] proposed LIPO, which embeds IPA in the generated binary with minimal overhead, and loads information about functions in other modules based on a profiling report. LIPO also removes the requirement of phony object dumps and link time wrappers when compared to LTO. Results showed a speedup of up to 10× in compilation time when compared to Feedback Driven Optimization (FDO) alone.

Following that, Johnson *et al.* proposes ThinLTO [16]. It differs from LTO by generating the function summaries on the LGEN stage for improved parallelism, while avoiding the large memory footprint requirement on the original LTO. ThinLTO also supports incremental builds by hashing the compiled function to avoid recompilation of unmodified functions, while LTO requires recompilation of the entire partition. Most GCC's IPA optimizations only use summaries to avoid this memory footprint issue.

3 INSPIRATION AND PROFILING

On a previous attempt to parallelize GCC, Bernardino *et al.* [4] presented a profiling report of how much time the compiler is consuming on each stage. We used the `-ftime-report` from GCC and compile the file named `gimple-match.c` to reproduce the profiling report that the authors used in their experiments, which we show on Figure 4.

Another way of doing this profiling process would be using an automated profiling tool like `gprof`[3] or `perf`[9] and identifying the

[3] https://sourceware.org/binutils/docs/gprof/

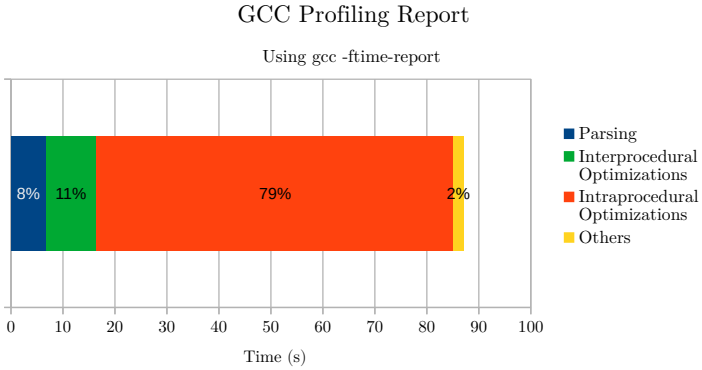

Figure 4: Profiling GCC with `gimple-match.c`

role of the labeled function in the program, but we chose not to use them once GCC had this autoprofiling functionality already implemented.

In GCC, all Intraprocedural Analysis are executed after IPA. When LTO is enabled, that analysis is done in the LGEN stage, which already runs in parallel when LTO is activated [14]. This inspired us to transplant the LTO partitioner for per-file classical compilation instead of threading the compiler, which resulted in a faster development process.

Therefore, the main idea is to partition the TU into multiple partitions, and compile them separately *after* the Interprocedural Optimizations were decided. Once each partition is compiled, we merge the compiled code into a single object file, maintaining compatibility with the original build tools.

3.1 Theoretical Maximum Speedup

On Figure 4, the compilation of `gimple-match.c` takes 87*s*, where 79% is spent on Intraprocedural Optimizations, consuming an relevant slice of the compilation time. If we assume that we can parallelize this part with linear speedup, and round up the time taken in the Intraprocedural part to 80% (which is $\frac{4}{5}$), then we can calculate the parallel time T_p with p processors as:

$$T_p = \frac{1}{5}T_1 + \frac{4}{5p}T_1 = \frac{1}{5}\left(1 + \frac{4}{p}\right)T_1$$

where T_1 is the serial time we measured. By the definition of speedup, the maximum speedup archivable in the compiler is:

$$\lim_{p\to+\infty}\frac{T_1}{T_p} = \lim_{p\to+\infty}\frac{T_1}{\frac{1}{5}\left(1+\frac{4}{p}\right)T_1} = \lim_{p\to+\infty}\frac{5}{1+\frac{4}{p}} = 5$$

Therefore, if we devise an mechanism to parallelize every Intraprocedural Optimization, we can theoretically archive up to 5× speedup when compiling individual files.

4 OUR PROPOSAL

We propose a modification in the LTO engine to make it work *without* having the context of the *entire* program. Once this is done, we can use the LTO partitioner engine to partition the file and compile it in parallel, similarly to LTO.

Our approach differs from LTO mainly in how we handle the IPA. LTO handles these optimizations with the context of the whole program, while our approach will only have the context of the original TU. This 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 will first discuss the internals of some parts of GCC, which we had to modify for our implementation to work. In Subsection 4.1, we present how we modified the GCC driver to reconstruct the output object file, required for maintaining compatibility building tools such as Make. Then, in Subsection 4.2 we present a short algorithm for making the LTO partitioner work for our proposal. In Subsection 4.3 we present a necessary change we had to do in our work about how partitions are applied in GCC. In Subsection 4.4, we explain how we solved the issue of private symbols with the same name being promoted to global. Then in Subsection 4.5, we present an optional 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 4.6.

4.1 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 another language B [2]. In GCC, it translates several languages to assembly of 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. 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 Figure 2.

Therefore, if we want our changes to not break the building scripts (*e.g.*, Makefile) used by most projects – which is mostly launching `gcc file.c -c` which creates 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 LTO *partitioner* to communicate the location of each created assembly file (.s) to the *driver*. This can be done by passing a hidden flag `-fadditional-asm=<file>` by the driver to the partitioner. 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 TU, it should *retarget* the destination assembly file and write the retargeted name to the communication file.
(2) Patch the driver to pass this hidden flag to the *partitioner*, and also to check if this file exists. If not, this means that (1) the compiler is incompatible or (2) it has chosen not to partition the TU. In the first case, the driver should call the assembler to every assembly file generated, and call the linker to generate the expected final object file. In the

second case, simply fallback to the previous compilation logic.

Figure 5 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, and the driver will launch the assembler (*as*) to generate several object files, to finally launch the linker (*ld*) to merge them all into a single object file.

4.2 Adapting the LTO Partitioner

In GCC, a TU 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, or there is a reference to a global variable g in f, then 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. Therefore, the LTO WPA engine must be further patched to expect that the function body is present when analyzing the program.

Then comes the partitioner algorithm *de facto*. In LTO, GCC tries to create partitions of similar size, and always try to keep adjacent nodes together. However, we devised an partitioning algorithm for our tests which only keeps together nodes which are *required* to be in the same partition. Therefore, we do not focus on partition balance, although we did implement a very simple partition merging logic for some balance.

This partitioning algorithm works as follows: for each node, we check if it is a member of a COMDAT [1] 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 group 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 of the group. Algorithm 1 presents a pseudocode of this process. Most of this process is already present from the COMDAT dependency algorithm from LTO. We added the logic to expand the partition frontier if a duplicated symbol is encountered, adding further non-duplicated symbols in to the boundary, as we illustrate in Figure 6. Notice how v_{16} is not added to the frontier, as is called by v_{14}, the symbol marked as duplicated.

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 a 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 [11].

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. We fixed that by checking if a marked to be removed function is inlined into a function inside the boundary. Second, being variables marked as in the boundary (and therefore not in the partition) not being correctly promoted to external. We further fix that by checking if there are references to that variable through other partition, and in this case we promote them to global.

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

4.3 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 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 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 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 assembly file to another file private to this partition and writes a pair (*partition number, path to file*) into 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 steps in the compiler does not emit assembly too early, such as were the case of the gimplifier in GCC, or else the output file will be incomplete. These cases could be easily detected by using a breakpoint and flushing the assembler output file before retargeting the new assembler output file.

Once the partition is applied to the child process, and the output file has been communicated to the driver, it can continue with the

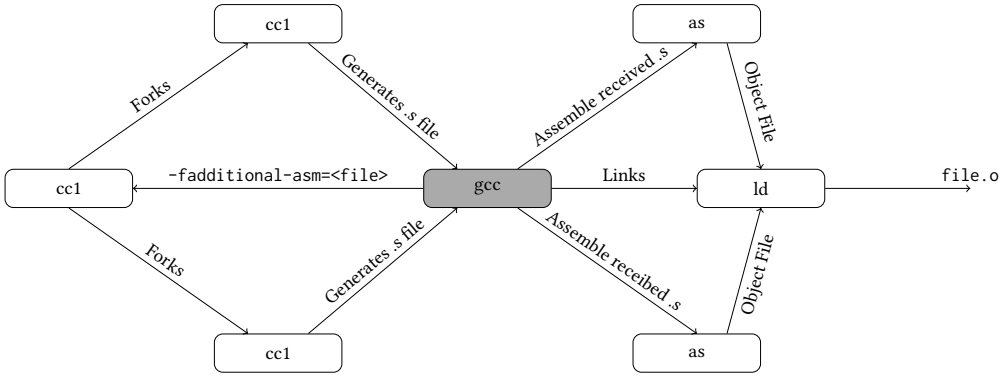

Figure 5: Interaction between the *driver*, the *compiler*, and other tools after our changes

Algorithm 1 Partition the callgraph into multiple partitions for parallel compilation

```
Partitions ds                                   ▷ Hold partitions. Can be implemented with Union-Find.
procedure LTO_MERGE_COMDAT_MAP(Callgraph G)     ▷ To be called by the LTO partitioner.
    Initialize ds   ▷ Initialize one partition per node. Partitions will be constructed by merging other partitions.
    for each symbol of G as v do
        if v belongs to a COMDAT group then
            merge_comdat_nodes(v, partition of v)
        merge_contained_symbols(v, partition of v)
    balance the partitions in ds
    build LTRANS partitions object from ds
procedure MERGE_COMDAT_NODES(node v, Partition set)   ▷ Merge the COMDAT group and its frontier.
    if v is marked then return
    mark v
    if v belongs to a COMDAT group then
        for every node in the same COMDAT group of v as w do
            unite partition of w with partition set
            merge_comdat_nodes(w, set)
    if v belongs to a COMDAT group or v is a duplicated symbol then
            ▷ If v is duplicated and not member of COMDAT group, we need to reach to an non-duplicated symbol to add to the frontier.
        if v is a function then
            for every caller of v as u do
                if calledge (u, v) is inlined or v is a duplicated symbol then   ▷ Same as above.
                    unite partition of u with partition set
                    merge_comdat_nodes(u, set)
                                ▷ thunk functions adjust their object `this` pointer before calling the function
            if v is a thunk function and v is not inlined anywhere then
                for every callee of v as w do
                    unite partition of u with partition set
                    merge_comdat_nodes(u, set)
                                ▷ Look at all nodes refering v, and add them to the COMDAT partition as well
        for every node referring v as u do
            unite partition of u with partition set
            merge_comdat_nodes(u, set)
            if u is a duplicated symbol then   ▷ If v is a duplicated symbol, we must reach an non-duplicated symbol for the frontier
                merge_comdat_nodes(u, set)
procedure MERGE_CONTAINED_SYMBOLS(node v, Partition set)
    for every node contained in v as u do
        unite partition of u with partition set
```

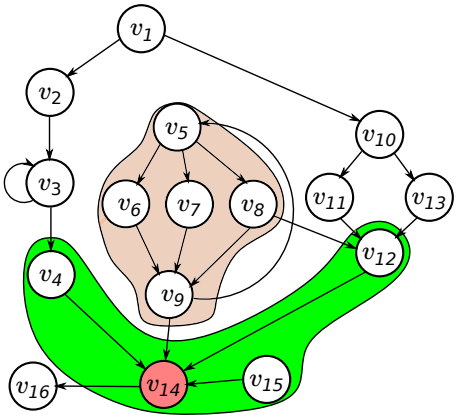

Figure 6: Example of callgraph, in beige being represented the COMDAT group, in green the COMDAT frontier, and in red the cloned nodes.

compilation. It should be running in parallel now by the number of partitions.

4.4 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 context of the entire program, which we do not. Therefore, we need another way of fixing it.

Two naive approaches would be to select a random integer and append to the function's name, or append the memory address of the function object to the function's name. Both of them breaks bootstrap [8], the first because output functions will have distinct names on every new compilation. The second because 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.

4.5 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 GNU Jobserver [7]. 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 TU 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 (*e.g.* `+gcc -c file.c`), and therefore it is not so straightforward to use this mode on existing projects.

4.6 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 from them the burden of this process. But nothing avoids that a malicious developer modifies the codebase *before* the distribution (*e.g.* inserting a backdoor), and claiming that he/her 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, for instance, to not contain some kind of build timestamp, and so on. 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 [17].

To keep compatibility with this project, we must ensure that our compiler always outputs 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:

- 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 certain 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 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 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.

5 METHODS

We ensure the correctness of our changes by (1) bootstrapping GCC; (2) ensure that the GCC testsuite was passing; and (3) compiling random programs generated with *csmith* [24] and ensure that the correct result was found. These tests were done with 2, 4, 8 and 64 parallel jobs, with minimum partitioning quota of 10^3 and 10^5 instructions.

For the time measurements, all points represent the average of collected samples, with errorbar 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. For the projects, we collected $n = 5$ samples because compilation is a computer intensive task. We made sure that in every test, `-g0` was passed for fairness, once in the current state we do not eliminate debug symbols that are assigned to other partitions, which if enabled imposes an unfair overhead due to several unnecessary writes to disk because of duplicated debug symbols, slowing down the parallel compilation process.

	No. Cores	No. Threads	RAM	Storage Dev.
Core-i7 8650U	4	8	8Gb	SSD
4x Opteron 6376	32	64	252Gb	HDD

Table 1: Machine specification of tests

Tests were mainly executed in two computers, which are represented in Table 1. The graphic caption specifies where the test was run. For the tests regarding the compilation of entire projects, we used the Opteron machine with `make -j64` as the baseline and then enabling 8 internal threads in the compiler. We then enabled the jobserver integration feature and let it use how many threads as GNU Make allowed.

The version of GCC used in the tests is available in `omitted`[4] with hash `450b006f3`.

6 RESULTS

We first highlight some of our results in Table 2. On this table, the autogenerated column means that this file is compiled to C++, and then compiled into assembly by GCC. Furthermore, Figure 7 shows the timings and speedups of compiling files with Number of Instructions (insns) $> 5 \times 10^4$. We used the insns as a metric instead of Lines of Code (LoC) as it better captures template-heavy code that has to generate multiple functions for each argument from the same input. In this figure, we can see that for large files, we mostly got speedups in general. We managed speedups of up to a 2.4× on Core-i7 when compiling individual files with 8 threads, and up to 3.53× on Opteron 6376 when with 64 threads. These speedup results are reasonably distant than the linear optimal (4.4 if $p = 32$), so these results could be further improved.

We also notice that there is a file with 0.7× speedup (*i.e.*, a slowdown) named `brig-lang.c`. The reason for that is the existence of a massive blob function that is generated when compiling this file (47261 insns, while the file itself has 55927 insns). This was not observed in the remaining files. Since we partition the file by functions and global variables, and this function consumes practically the entirety of compilation time, we were not able to extract any meaningful parallelism here.

We will now discuss how our proposed changes impact the overall compilation time of some projects. The objective is to check if this parallel file compilation mechanism generalizes to other projects that do not have those massive blob files, as in GCC. For this, we run experiments compiling the Linux Kernel 5.8-rc6, Git 2.30.0, the GCC version mentioned in Section 5 with and without bootstrap enabled, and JSON C++, with commit hash `01f6b2e7418537`. We selected those projects as they are very often used on the Open Source community. We have only enabled jobserver integration in GCC and Git, because it is necessary to modify an absolute large number of Makefiles for that (for instance, Linux has 2597 Makefiles).

In Figure 8 we show our results. We ran this benchmark on a 64-threaded Opteron machine instead of the Quad-Core i7 machine, once we find reasonable that our methodology would have better results on high core count machines, and close to none improvement on low-core count machines. 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 also squeezed a small improvement in GCC compilation, but showed a massive slowdown in Git. This is because Jobserver integration has interprocess communication cost with Make, which is a problem if the size of the partitions is small. Other than this, we seen no significant speedup or slowdown in these other projects.

We first analyze why GCC had such a 35% speedup. In Figure 9, we plot the interval graph of each file again, but in this time enabling threads inside the compiler. We can observe a significant reduction in both file compilation time and the parallelism bottleneck. This is due to our parallel file compilation mechanism.

Now, this 15% improvement on Git was not expected at first, which was an excitement. Analyzing the compilation process of git, we found that it simply did not have enough files to fully populate the 64-threaded Opteron machine. Therefore, the extra parallelism added by our method improved the CPU usage here. However, this project compiles quite fast in this machine (5*s*), so the improvement is not much noticeable in practice.

However, on the Linux kernel, we see no improvement nor slowdown by our technique. The reason is that the files in this project compile very fast, and it is evenly distributed through the processors' cores. The same thing applied to the JSON C++ project.

Another interesting observation from both Figures 1 and 9 is the high amount of serial steps when compiling GCC until the high throughput compilation phase starts. This can be attributed to GNU Autotools configuring the compiler for the environment of the machine. Perhaps the configure part could be parallelized as well, and this could improve several projects across the board, or replace this tool altogether with another that can better handle parallelism.

7 CONCLUSIONS, ISSUES, 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.

Our main conclusion is that parallel file compilation is useful in two scenarios: (1) in the existence of massive autogenerated blob files in the the project, which consumes a large amount of compilation time when compared to the remaining files of the project, and (2) if the number of files in the project is not enough to fully populate the processors of the machine. This means that our approach has some generalization limitations, but can result in speedups. Else, our methodology performs as well as the default per-file compilation parallelism. Furthermore, if this is implemented as a feature in a compiler, it could be used in projects where LTO imposes a slowdown. Johnson *et al.* show some of these projects when presenting ThinLTO [16].

Furthermore, there are several points in which our work can be improved. Mainly, the issue with our partitioner applier not removing debug symbols associated with removed nodes, resulting in duplicated symbols being dumped into the final assembly. This is a programming issue rather than a design flaw, and therefore it is still fine as a proof of concept to support our claims.

[4] omitted according to the double-blind policy

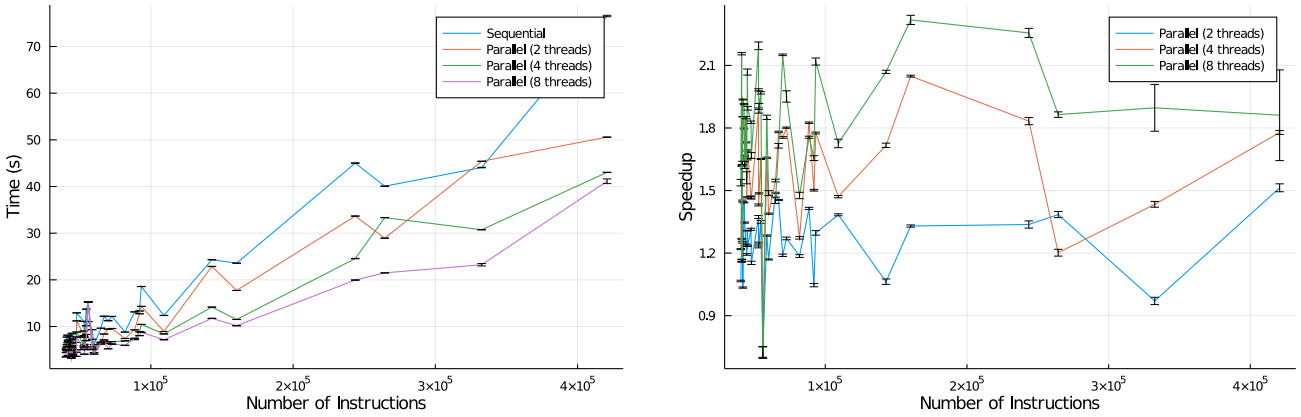

Figure 7: Times and Speedups of selected files with 1, 2, 4 and 8 threads on Core-i7

	Core-i7			Opteron 6376				
File	Sequential	8 Threads	Speedup	Sequential	64 Threads	Speedup	Autogenerated	LoC
gimple-match.c	76*s*	31*s*	2.4×	221*s*	66*s*	3.32×	yes	244320
insn-emit.c	23*s*	10*s*	2.25×	97*s*	37*s*	3.53×	yes	273482
tree-vect-stmt.c	11*s*	5*s*	2.14×	32*s*	13*s*	2.46×	no	12133/
brig-lang.c	10*s*	15*s*	0.7×	29*s*	35*s*	0.8×	no	958

Table 2: Speedup of highlighted files

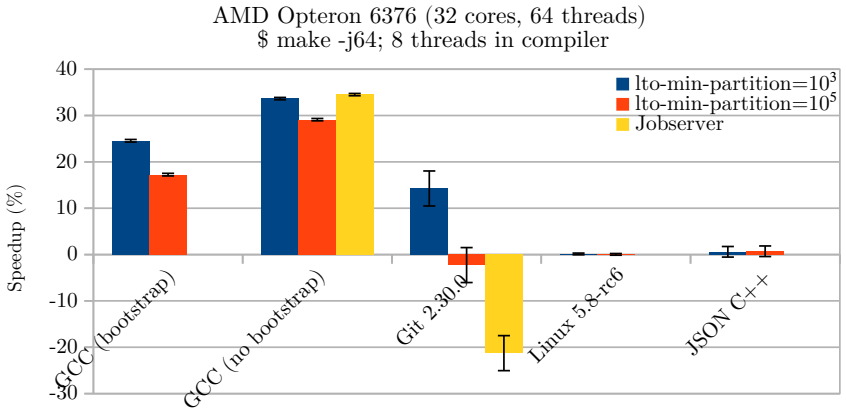

Figure 8: Compilation of some projects with 64 Makefile jobs and 8 threads in compiler in Opteron 6376

Second is by implementing a better load balancing algorithm to the project, once we only kept its load balancing algorithm minimal. Using the LTO default partitioner as a base is a good start.

Third, try to develop a predictive model to decide if the input file is a good candidate for parallel compilation. Figure 7 shows a clear linear correlation between the expected number of instructions and time, but it may be possible to use other informations as well.

And fourth, try to avoid partial linking and symbol promotion altogether by concatenating the generated assembly files, instead of linking every generated assembly file into temporary object files. This should eliminate any possibility that promoting symbols to public visibility may impact the binary performance in a negative way.

We also would like to draw attention to LTO's WPA step, which still runs sequentially. Parallelizing this step, which includes all IPA 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 79% of the compiler.

REFERENCES

[1] 2021. Whole program assumptions, linker plugin and symbol visibilities. https://gcc.gnu.org/onlinedocs/gccint/WHOPR.html
[2] Alfred V. Aho, Ravi Sethi, and D. Jeffrey Ullman. 2007. Compilers, Principles, Techniques, and Tools. Second Edition.
[3] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. 2015. Parallel Parsing Made Practical. *Sci. Comput. Program.*

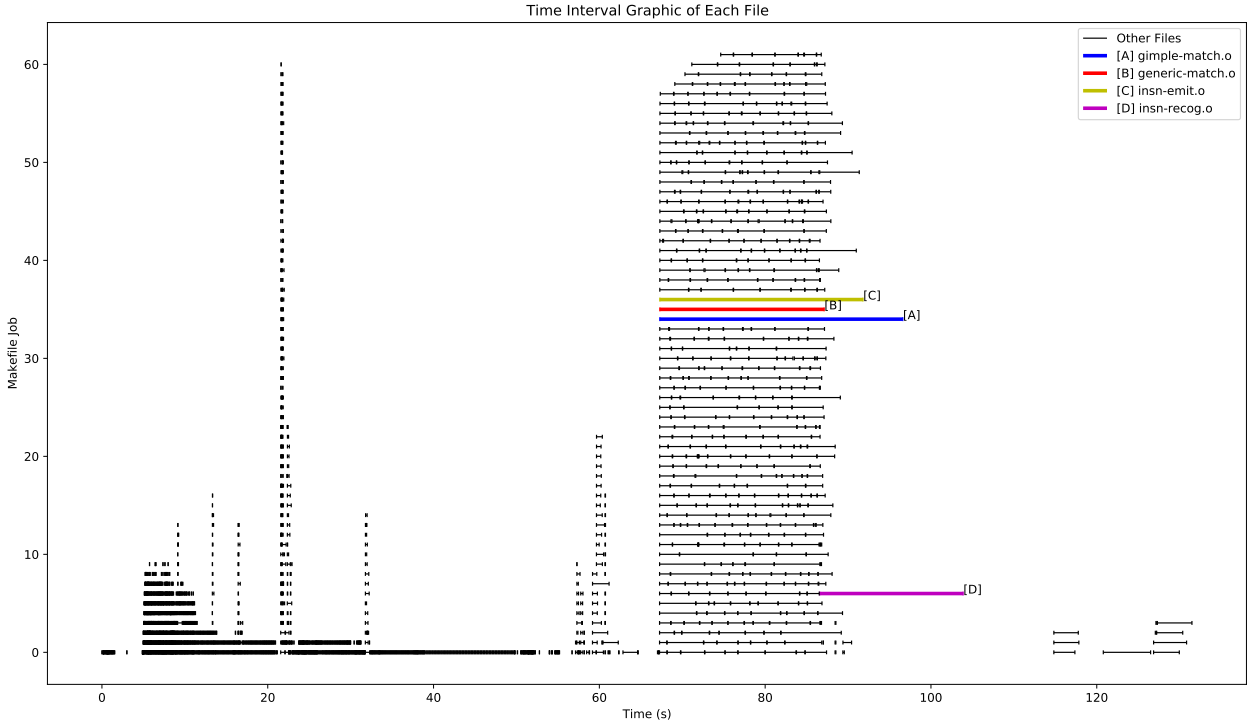

Figure 9: Compilation time of GCC, using 8 internal threads and partition if *Number of Instructions* $> 10^3$

112, P3 (Nov. 2015), 195–226.

[4] Matheus Tavares Bernardino, Giuliano Belinassi, Paulo Meirelles, Eduardo Martins Guerra, and Alfredo Goldman. 2020. Improving Parallelism in Git and GCC: Strategies, Difficulties, and Lessons Learned. *IEEE Software* (2020).

[5] Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, and Ollie Wild. 2007. WHOPR - Fast and Scalable Whole Program Optimizations in GCC. https://gcc.gnu.org/projects/lto/whopr.pdf

[6] GNU Collaborators. 2005. Link-Time Optimization in GCC: Requirements and High-Level Design. https://gcc.gnu.org/projects/lto/lto.pdf

[7] GNU Collaborators. 2020. Posix Jobserver Interaction. https://www.gnu.org/software/make/manual/html_node/POSIX-Jobserver.html

[8] GNU Collaborators. 2021. Installing GCC: building. https://gcc.gnu.org/install/build.html

[9] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[10] Patrick Dubroy and Alessandro Warth. 2017. Incremental packrat parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. 14–25.

[11] Paulo Feufiloff. 2002. Union-Find. https://www.ime.usp.br/~pf/mac0122-2002/aulas/union-find.html

[12] Charles N. Fischer. 1975. *On parsing context free languages in parallel environments.* Technical Report. Cornell University.

[13] Seth Fowler and Joshua Paul. 2009. Parallel Parsing: The Earley and Packrat Algorithms. (2009).

[14] T. Glek and Jan Hubička. 2010. Optimizing real world applications with GCC Link Time Optimization. *GCC Developers' Summit Proceedings* (2010), 25–46.

[15] GNU. 2019. GCC Releases History. https://www.gnu.org/software/gcc/releases.html

[16] Teresa Johnson, Mehdi Amini, and Xinliang David Li. 2017. ThinLTO: scalable and incremental LTO. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 111–121.

[17] Chris Lamb, Valerie R. Young, and Clemens Lang. 2016. When is a build reproducible? https://reproducible-builds.org/docs/definition/

[18] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[19] David Xinliang Li, Raksit Ashok, and Robert Hundt. 2010. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 53–61.

[20] Neil Lincoln. 1970. Parallel Programming Techniques for Compilers. *SIGPLAN Not.* 5, 10 (Oct. 1970), 18–31.

[21] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning BOLT: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 119–130.

[22] Mark Thierry Vandevoorde. 1988. *Parallel compilation on a tightly coupled multiprocessor.* Systems Research Center.

[23] David B. Wortman and Michael D. Junkin. 1992. A Concurrent Compiler for Modula-2+. *SIGPLAN Not.* 27, 7 (1992), 68–81.

[24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.