FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

**BACHELOR THESIS**

Michal Jireš

# Incremental link-time optimization in GNU Compiler Collection

Department of Applied Mathematics

| | |
|---|---|
| Supervisor of the bachelor thesis: | doc. Mgr. Jan Hubička, Ph.D. |
| Study programme: | Computer Science |
| Study branch: | System Programming |

Prague 2023

I would like to thank my supervisor Jan Hubička for great guidance during development and writing of this thesis.

Title: Incremental link-time optimization in GNU Compiler Collection

Author: Michal Jireš

Department: Department of Applied Mathematics

Supervisor: doc. Mgr. Jan Hubička, Ph.D., Department of Applied Mathematics

Abstract: Modern compilers attempt to optimize programs as much as possible. One such significant effort is Link–Time Optimization (LTO). LTO takes the whole program as accessible to a linker, and performs global optimizations that are impossible in previously local compilations. Because of the global nature, LTO must be performed in full in each compilation, which results in long compile times even in edit–compile cycles. Incremental compilation can reduce compile times of edit–compile cycles by reusing unchanged objects.

This thesis aims to implement incremental compilation for Link–Time Optimizations of GNU Compiler Collection, specifically of local transformation phase. We implement incremental compilation by caching files of compilation units of local transformation.

For best success of incremental compilation we also aim to minimize number of changed compilation units after small edit. We achieve this in two ways. First, we create better partitioning strategy, that concentrates the changes into fewer compilation units. Second, we analyzed sources of divergence and, if easily possible, removed them. That includes stabilizing values and fixing streaming and inter–procedural optimizations to increase their robustness against small edits. Both without influencing quality of the final binary.

Keywords: compiler, optimization, link–time optimization

# Contents

# Introduction

Modern compilers provide increasing amount of optimizations. Traditionally optimizations are performed only locally in code corresponding to a single source file, but for last 2 decades optimizations across source files are becoming common. Cross source files optimizations may provide direct benefit of performance, smaller code size, and better warnings. They also have indirect benefit of requiring less strict decomposition into source files to achieve best performance.

Compilers such as GNU Compiler Collection and LLVM implement cross source file optimizations using Link–Time Optimizations (LTO). Some Linux distributions, such as SUSE since 2019[1], already turn LTO on by default for their packages. With such widespread adoption, it is important to make the development with LTO enabled as painless as possible. While it is possible to develop with LTO disabled and only use LTO for the final binary, their behavior may not be entirely identical, be it in performance or because of bugs. Thus it is desirable to reduce compile times of LTO for edit–compile cycles with typically minor changes. Such reduction can be achieved by incremental compilation, which reuses unchanged results of local transformation units from previous edit–compile cycle. This is already implemented in LLVM.

This thesis aims to implement such incremental compilation in GNU Compiler Collection. In addition we analyze sources of divergence which make reuse impossible, in many cases even without influencing the quality of the final binary. And in such cases we attempt to remove these sources by stabilizing values and by making some algorithms in GCC more robust against small changes.

Section 1 introduces the reader into the world of compilers ending with explanation how LTO works in GCC and quick overview of alternative approaches LIPO and LLVM's ThinLTO. Section 2 describes how we implement the incremental compilation with cache. In Section 3 we design partitioning strategy that partitions local transformation units to minimize propagation of divergence. In Section 4 we analyze the sources of divergence and attempt to fix them. In Section 5 we explain how to apply our changes to GCC and how to use them. In Section 6 we measure how successful is our implementation in reusing results of local units.

[1]https://lists.opensuse.org/archives/list/factory@lists.opensuse.org/message/UT2YVWPZK2IZ5EUHMSHNCW3Q72CMPWCJ/

# 1. Introduction to Compilers and Link–Time Optimization

## 1.1 Compilers

Compilers are programs that translate program description from one language to another [Muc+97]. Most common example is to translate from a programming language, such as C, to machine code of target architecture. However this translation is often not done directly, and there exist multiple intermediate languages.

We will be working on GNU Compiler Collection[1] (GCC). GCC is open source compiler supporting many programming languages such as C, C++, and Fortran, and many target architectures. It is the primary compiler used in Linux ecosystem.

## 1.2 Optimizations

Important feature of modern compilers are optimizations. In a language there may exist multiple ways to describe an identical program. Compilers try to find such transformations that the observable behavior of a valid program is identical, while the program requirements improve — such as faster execution or smaller binary size.

Optimizations are usually done on intermediate languages designed for easier and faster modification useful for given optimization strategy. Following is a list of examples of simple transformations:

- Dead code elimination removes parts of the program that will never be used/executed [Muc+97, p. 592]. Trivial examples would be unused variable or `if(false)` statement. Such a statement might not be common in code written directly by a programmer, but it is common result of other optimizations.
- In most programs there are variables that do not change their value during any execution of the program. During constant propagation compiler identifies such variables and replaces their usage with a constant [Muc+97, p. 362]. This then allows other passes such as dead code elimination.
- Inlining replaces call to a function by copy of the function's body [Muc+97, p. 465]. This removes overhead required to call a function, which may be significant for trivial functions. However the more important result of inlining is that it allows optimizations across, now nonexistent, function boundary.

Modern compilers contain hundreds of such optimization passes. We will focus on inter-procedural optimizations. List of such optimizations is in Section 1.5.3.

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

## 1.3 Standard compilation

Large programs consist of multiple source files. In standard compilation each source file is compiled into machine code as one compilation unit with no interaction with other source files. Results of these compilation units are represented as object files. Object files contain symbols which can represent for example functions, variables, and debug information. Object files are then glued together by a linker. This process is shown in Figure 1.1.

Since the compilation units are unrelated, this process can be trivially parallelized. It also allows trivial incremental compilation, by simply reusing object files from previous compilation as long as the corresponding source file has not changed.

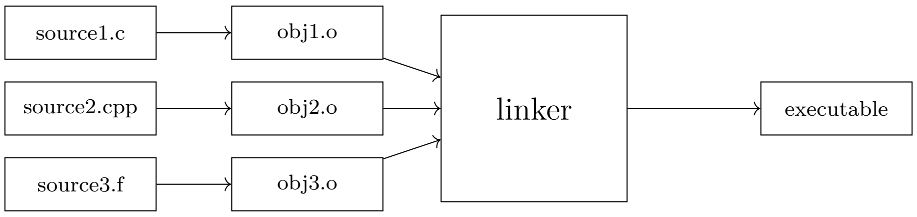

Figure 1.1: Standard compilation

## 1.4 Link–Time Optimization (LTO)

While the standard compilation has many benefits given its simplicity, it also misses many opportunities to optimize the code across source files. These unrealized opportunities to optimize across source files is the target of Link–Time Optimization (LTO). It also allows us to optimize across different programming languages.

As the name suggest these further optimizations are done during linking where we can see the whole program at once. Initial compilation is similar to the standard compilation and covers the local optimizations. However because the program will be further optimized, the object files now must contain intermediate language instead of final machine code.

LTO is not a new concept. In 1990 there was attempt to implement LTO directly as part of linker, called loader in Plan 9 [Tho90].

## 1.5 GCC LTO

LTO in GCC was designed [05; Bri+07] and implemented [GH10] in years 2005–2010 and since 2019 it is in some Linux distributions turned on by default[2]. First was LTO mode [05] which reads the whole program at link–time and optimizes it as if it was single compilation unit, which is not parallelized. Later came WHOPR mode [Bri+07] which divides link–time optimizations into hard to be parallelized global phase and easily parallelizable local units phase which should do most of

[2]https://www.phoronix.com/news/OpenSUSE-Tumbleweed-LTO

the work. This leads to shorter compile times and less memory usage than LTO mode. We will talk only about WHOPR mode of LTO and call it simply LTO.

We represent the program using a callgraph (with nodes being functions and edges being call between them). We do so because inter–procedural passes, the main focus of LTO, depend on call relations of functions which is well represented by the callgraph.

LTO in GCC consists of 3 main stages: [GH10, p. 3]

- Local generation (LGEN) happens in place of compilation units of standard compilation and can be identically parallelized. LGEN mimics compilation units of standard compilation, so that they can share large parts of implementation, and so LTO can be enabled by single `-flto` flag without changing building process. The source files are compiled into intermediate language and packed into `lgen` object files together with their summary information.

- Whole program analysis (WPA) is the only phase which works with the whole program and thus is difficult to parallelize. It decides what transformations should be made, but it only modifies the summaries. The transformation itself is postponed into later parallelizable stage. At the end it partitions the global callgraph into several local callgraphs into `ltrans.in` object files, one for each LTRANS unit.

- Local transformation (LTRANS) transforms the code from `ltrans.in` according to optimization summaries from WPA into `ltrans.out`.

The basic relations of these stages and linker is shown in Figure 1.2.

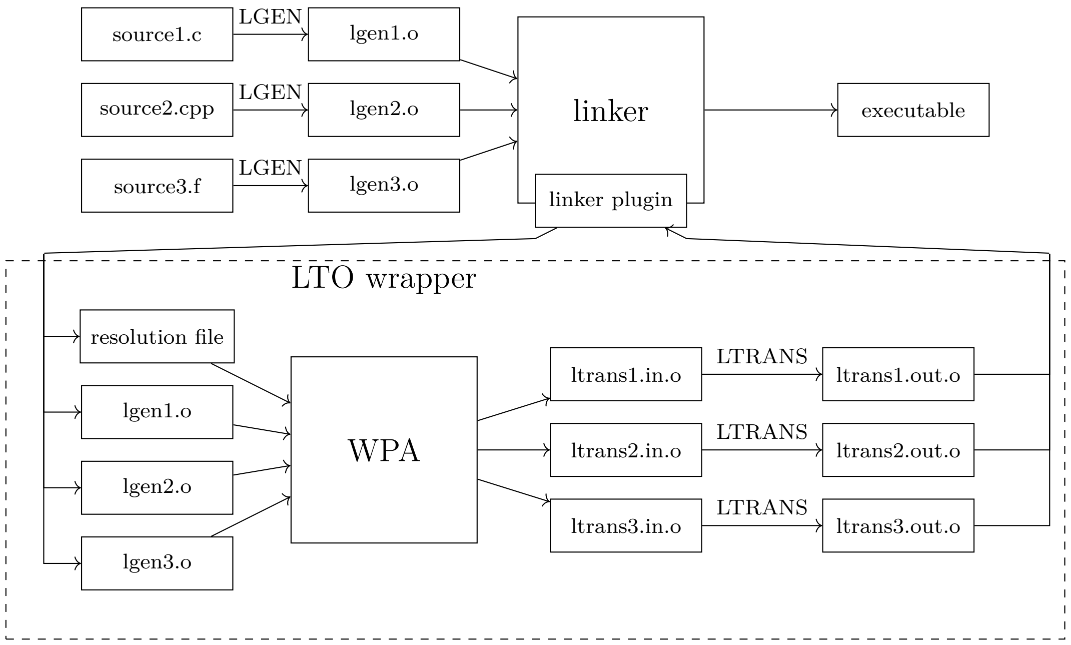

Figure 1.2: GCC LTO compilation

### 1.5.1 Local generation (LGEN)

LGEN compiles source files into intermediate language. LGEN performs early optimizations, with inter–procedural optimizations being limited by local knowledge. While these optimizations could be performed in later stages, this phase is easiest to reuse and thus it is preferable to keep the work in early stage [GH10, p. 3]. Implementation of these optimizations is shared with standard compilation.

Function bodies are analysed and relevant information is streamed into object files as summary, along with the intermediate code.

### 1.5.2 Linker plugin — LTO wrapper

GCC inserts itself into linker using linker plugin [GH10, p. 6]. Plugins are not specific to GCC and are supported by multiple linkers. Such a plugin can recognize object files compiled with LGEN, and replace them with LTO optimized object files. It also gains information how the symbols are resolved which is written into resolution file. The resolution information specifies how the symbols will be linked together, and from it we can find out whether symbols are exclusive to our LTO context or they are externally visible by for example a shared library which we do not optimize with LTO.

The linker plugin sends the recognized object files to `lto_wrapper` which executes WPA and LTRANS stages and returns back LTO optimized files.

### 1.5.3 Whole program analysis (WPA)

WPA takes all the object files and tries to find possible optimizations between them. WPA contains many passes, that may become relevant to our caching effort. Chronologically:

1. *Tree merging* streams in types and declarations from object files and unifies duplicates.
2. *Linking* creates a global callgraph according to resolution file.
3. *Summary based optimizations*:
   (a) *Symbol promotion* takes global symbols and if possible localizes them by making them static. This then allows local reasoning about the symbol. For example it can remove the symbol if it is inlined in all locations.
   (b) *Profile propagation.* Profiles, either measured or estimated, specify how often is each part of the callgraph expected to be used. This pass propagates these estimates across function boundaries.
   (c) *Identical code folding* finds identical duplicates of functions and merges them [Liš14]. Such duplicates may be a result of templates with similar types.
   (d) *Devirtualization* turns polymorphic calls using virtual functions into direct calls [Hub14]. In clearest example, if virtual function is implemented only by one class, we can call that single implementation directly.

   (e) *Iter–Procedural Constant propagation* analyses arguments of functions to find which are constant. It also can create cloned versions of functions which specialize for a given constant [LN05].
   (f) *Scalar Replacement of Aggregates* optimizes functions arguments by replacing an aggregate, such as struct, with only its contents. This is useful for example when only single struct member is used inside the function [Jam09].
   (g) *Constructor merging* creates a function that calls all static constructors. In ideal circumstance the call will be inlined and we will have a single function initializing the whole program [GH10, p. 9]. This way the constructors are together, and we reduce how much of the binary must be read to execute static constructors at the start of the program execution.
   (h) *Inlining*, as described earlier, inserts function body into caller instead of the call. During WPA it has much more useful global information and tries to inline as much as usefully possible within its parameters [GH10, p. 9]. Inlining across source files is among the main benefits of LTO. It is also the most time consuming pass of WPA.
   (i) *Function attribute discovery* auto–detects optimization hints in form of attributes, such as `noreturn`, `const` [GH10, p. 7, 10].
   (j) *Mod–Ref* and *Reference* passes identify which memory locations (function arguments, static variables) can be modified or referenced by any given function [Hub21]. It is useful for preventing unnecessary reads after calling a function that does not access the value.

4. After optimizations global callgraph is partitioned into partitions, each corresponding to one compilation unit during LTRANS phase.

5. After partitioning we need to fix static symbols. If original LGEN unit is split into multiple partitions, and at least 2 of those partitions interact with the same static symbol, the symbol must be promoted to be global. If a global symbol with the same name already exists, it must be also renamed. If two LGEN units are partitioned together and both have identically named static symbol, their names must be changed to be unique.

6. Finally, symbols and optimization summaries of each partition are streamed out into corresponding `ltrans.in` file.

### 1.5.4 Local transformation (LTRANS)

Local transformation phase works on local compilation units which can be easily parallelized. Each LTRANS unit transforms the callgraph based on optimization summaries from WPA phase.

They start with `ltrans.in` object files containing intermediate language and end with `ltrans.out` object files containing machine code, which are fed back to linker.

### 1.5.5 Early debug

In LTO debug information is handled separately from WPA and LTRANS phases using early debug [Bie15]. LGEN creates debug information and streams it into `lgen` file. However it is not linked directly with any declarations. Instead symbol of LGEN compile unit and offset into it's debug symbols is used to identify debug information in WPA and LTRANS.

When building `ltrans.out`, LTRANS adds shadow early debug with the symbol and offset pair, which during final linking will be replaced with corresponding debug information by linker.

## 1.6 Alternative approaches

It is good to mention that there exist multiple different approaches to cross source file optimizations.

### 1.6.1 LIPO

*Lightweight feedback-directed cross-module optimization* (LIPO) [LAH10] achieves cross source file optimizations without use of LTO. Instead it first requires measured profile specifying which function calls are used the most for typical inputs. With the profile the compilation starts again, each source file having its individual compilation unit. However depending on the profile, the source file is temporally joined with another source files to allow inlining of likely candidates. After inlining phase, everything from the other joined file is thrown away.

This has benefit that it is easily parallelizable, does not require any special intermediate language or support from linker, but there needs to be support from front–end to be able to combine source files.

The negatives are that it requires profiling, does not mix languages well. Also because there is no phase with global knowledge, some optimization passes are impossible.

LIPO was implemented for GCC by Google, but it never became official part of GCC and is no longer maintained. Relatively few users use profile feedback required for LIPO and standard LTO became the focus.

### 1.6.2 ThinLTO

LLVM[3] is another open source set of compilers which is the major competitor to GCC. It contains ThinLTO [JAL17], LTO implementation that aims to be highly parallelizable.

ThinLTO is similar to GCC in the structure of phases, but is inspired by LIPO to further thin out the global phase even at cost of optimization opportunities. It uses similar 3 phases:

- *Compile* phase is principally identical to LGEN.
- *Thin link* is a serial global phase similar to GCC's WPA with the main difference that it only works purely with summaries. While GCC's WPA

[3]https://llvm.org/

also loads the intermediate representation from object files and partitions them. It is intended to be as minimal as possible, and does not do any complex passes.

- *ThinLTO backend* is parallelizable phase similar to LTRANS. However each ThinLTO backend corresponds to a single Compile unit and starts with object file of that unit. The ThinLTO backend reads the summaries and transforms the code according to them. However, because there was no partitioning step, each unit must import symbols required for these transformations by itself.

Because of the more decentralized nature it is easier to parallelize, but some optimizations are not possible during local importing instead of global partitioning. This leads to worse and larger generated code.

ThinLTO implements incremental strategy very similar to the one we created in this thesis. Both ThinLTO and our implementation use cache on disk using hash for lookup, to skip compilation of given LTRANS unit. ThinLTO computes the hash of object file from Compile phase, which is identical to object file from backend phase, and of the summaries. GCC implementation combines these two objects into `ltrans.in` file, containing both the symbols for LTRANS phase and the summaries, of which our implementation takes the hash. The major difference is that `ltrans.in` can contain much more information created during WPA, and thus there are more possibilities for divergence, intended or not.

# 2. Incremental Compilation - Cache

## 2.1 Incremental Compilation

Incremental compilation is a compilation that reuses results from previous compilation runs. It is useful to reduce needless work and thus reduce edit–compile cycle times.

With standard non–LTO compilation, compilation units are local and don't influence each other. So we only need to recompile changed source files. In typical case, the developer changes a single source file, in which case only a single file must be recompiled leading to fast edit–compile cycles. Build systems, such as *make*, can simply check which source files were modified after the last compilation.

With LTO `ltrans.in` files are recreated during each compilation and can be potentially influenced by any source file. However it still holds, that if `ltrans.in` is identical, `ltrans.out` will be identical as well. It is also reasonable to assume, that small changes in source files will change only small amount of code at LTRANS stage. With good partitioning the changes may be localized to few partitions and many `ltrans.in` files will be identical to the ones in previous compilation.

To achieve incremental compilation, we implement cache around LTRANS phase, as suggested in Figure 2.1, which holds `ltrans.*` file pairs and will compare new `ltrans.in` files against them and in case of cache hit, will skip LTRANS compilation.

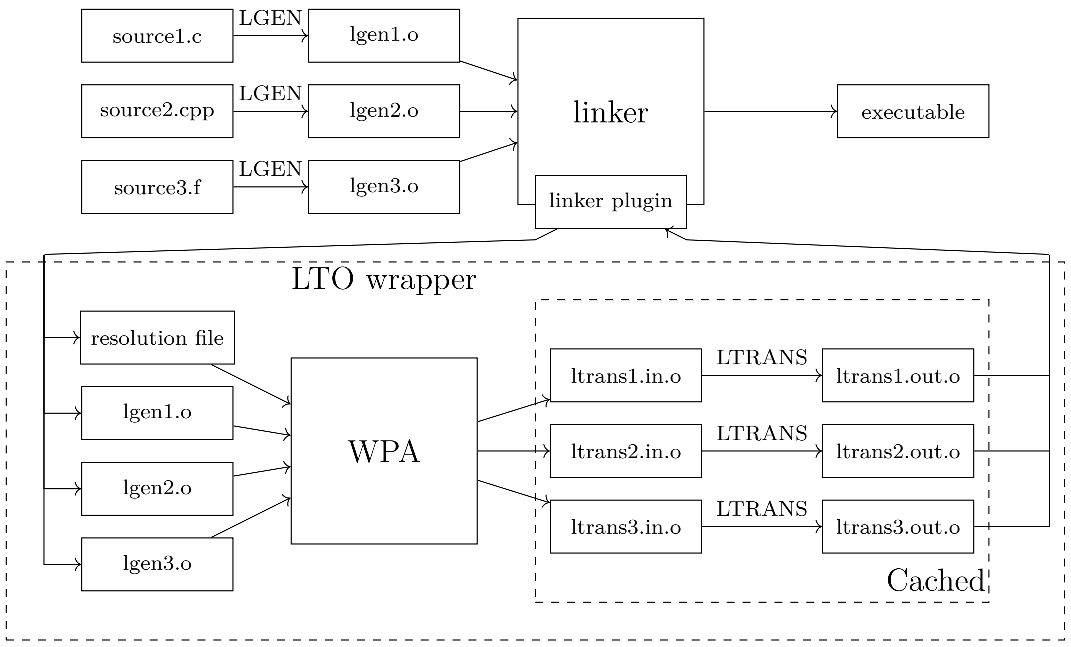

Figure 2.1: GCC LTO compilation with cache

## 2.2 Time spent during WPA vs LTRANS

Before we implement the cache, it is needed to establish that LTRANS takes significant amount of time, so that cache can have any impact. We can exclude LGEN phase, because it's incremental compilation is trivial just as standard compilation. Thus we compare only WPA with LTRANS.

LTRANS by design is supposed to do majority of work. However it is also parallelizable while WPA is not.

For attachment A.2 we tried to self compile GCC's C++ compiler `cc1plus` on machine with 24 threads. The WPA took 13s and in total real time and whole LTO took 52s. Thus even on such relatively parallelized machine the LTRANS is the bottleneck of LTO. Thus the cache would be useful to decrease the LTO compile time.

Furthermore, in the attachment there is also measured time spent in individual passes of WPA. It is dominated by two passes: streaming in (14 %) and inlining (42 %). If WPA becomes the bottleneck, they have a potential to be improved. GCC developers work on improving WPA compile times, continually improving it. There are also plans to parallelize WPA with threads but it is a complicated problem.

## 2.3 Requirements

Primary purpose of cache is to reuse previously compiled LTRANS partitions. Identical and only identical `ltrans.in` files if found in cache will prevent compilation of this partition and cached `ltrans.out` will be taken.

This should be achieved with low overhead, we will first compare by checksum and only then we will try to compare `ltrans.in` files byte by byte.

The cache should be parallelizable. This is needed in case of current build systems which build multiple executables at the same time. With parallelization we can simply specify cache globally without changing anything in the build system. Second reason is potential future use to reuse partitions across similar binaries, e.g. GCC's C and C++ compilers.

The cache should have a way to remove longest unused cache items, so that cache size cannot grow infinitely.

## 2.4 Design

Whole cache implementation fits into `lto-wrapper` which is called by linker and calls WPA and then LTRANS stages.

Relevant patches from A.1 are 3rd and 4th which implement filelock and the cache itself respectively.

### 2.4.1 Representation

Environmental variable `GCC_LTRANS_CACHE` specifies directory where the cache is located. If the variable is not set, or is not a valid directory the cache is ignored.

The persistent cache info contains only `ltrans.*` pair filenames, checksum of `ltrans.in` file to quickly find cache hits, and counter of last usage for each cache info. Temporary cache info also includes maps from `ltrans.in` filename / checksum to cache item.

### 2.4.2 Caching process inside lto–wrapper

First we can execute WPA phase mostly as normal. However since the resulting `ltrans.in` files will likely become part of cache, we need these files to be on the same disk partition to allow quick move to a new location. We achieve this simply by instructing WPA to create files in cache directory with some prefix. The prefix needs to be unique, so that we can run multiple WPA phases at once, uniqueness is achieved using `mkstemps`.

After that we setup cache and insert new `ltrans.in` files, cached `ltrans.*` filenames will be given by cache. If the pair already existed in cache, the new `ltrans.in` file is deleted. If the pair was not in cache, the new `ltrans.in` file is moved into cached `ltrans.in` filenames, and we are expected to fill `ltrans.out` filename by the new compiled file. Then we update cache directory to include new cache items.

We now need to compile all LTRANS partitions that were not found in cache. This works mostly as if there was no cache. However without cache, `ltrans.in` files are deleted once compilation of given partition is completed, while with cache we need to keep them.

Finally we have all LTRANS partitions compiled and cache filled and updated. We need to return these files to the linker, but the linker will delete them, so we create copy of all returned files using `link` to prevent deletion from cache.

### 2.4.3 Pruning

The cache automatically deletes cache items that were unused for the longest time. To know which cache item was longest unused, each item contains value of last usage, which is written from global counter whenever some cache item is used.

When cache is initialized from cache directory, items' last usage is remapped to lowest natural numbers while conserving their usage order.

If the cache has too many items, the cache is automatically pruned at the end of `lto-wrapper` once everything else is completed. The longest unused items are deleted until cache size fits into the limit.

## 2.5 Synchronization

Synchronization is required to allow parallelization. This implementation does not require locks across neither WPA nor whole LTRANS phase. Only when pruning is forced, we need to wait for all other accesses to finish.

### 2.5.1 Filelock implementation

Filelocks are now implemented using `fcntl`.

Filelock implementation only needs to implement write-lock and shared read-lock for cache to function. Which should be available on any platform LTO is likely to be used on.

### 2.5.2 Locks Used

Three different lock types are used to synchronize.

- First is a lock for creating new items in cache. This is always write–lock, only one process can create new items.
- Second are locks for individual cache items. If new cache item is created, we need to prevent other processes to use this item until the `ltrans.out` file is compiled. These locks are never directly on the files related to them, but they are on files with added suffix. This is because `ltrans.out` are handled by different subprocesses and the files can be created/deleted by them, or filelock implementation could prevent those subprocesses to use locked files.
- Final one is global deletion lock, that is only needed for deleting items. This lock is set as read–lock by any process that needs cache items to persist.

### 2.5.3 Lock Usage

Before and during WPA phase, no lock is needed.

During insertion of new `ltrans.in` files, we need to hold write-lock for creating new items. This way we prevent two processes allocating cache item with identical name. For all cache items that were not already in cache, we write-lock them, so that no other process can access them until they are compiled. After updating cache directory, we can release the lock for creating new items.

For all files that we compile, we already hold a write–lock, so LTRANS phase compiling is synchronized where needed.

To create copies of `ltrans.out` files for linker, we lock the given cache item with a read–lock to make sure that compilation of `ltrans.out` file is finished. If we already hold write–lock we relock as read–lock, this does not need to be atomic. After creating copies for linker, all locks can be released.

During all previous steps since before creating new files is locked, read–lock to deletion lock is held. Now we can try to write–lock deletion lock to prune the cache from long unused items.

If deletion is important to happen more frequently it would be possible prune without a global lock and more granularly. Though it would require a second lock for each file pair, to signify whether the pair is intended to be used.

### 2.5.4 Impossibility of deadlocks

Deadlocks makes execution of a process impossible because one process while holding lock `A`, tries to lock `B`, while second process holds lock `B` and unlocking of `B` is in some cyclical nature dependent on `A` being unlocked.

We prevent deadlocks for each individual lock:

- During holding of a creation lock, only locks on newly created files are created, which cannot be locked by any other processes.
- While holding locks to individual files, only other locks to individual files can be locked. Since the we try to read–lock, only previous write–lock can prevent us locking. Write–locks are only created during creation, read–locks depend only on files that were already in cache during creation, and each compilation has only single creation. Thus there cannot be a cyclical dependence.
- Active deletion write–lock is mutually exclusive with (trying to) holding any other locks and its creation is optional.

# 3. Partitioning Strategy

WPA phase partitions symbols into partitions — `ltrans.in` files. Each partition corresponds to a single LTRANS unit which can be cached. For cache to be successful for a given partition, contents of that partition cannot change from previous edit–compile cycle.

Partition can change for multiple reasons:

- Symbols can change. This is mostly covered by next chapter. However if symbol change is inevitable we can minimize number of changed partitions by concentrating changed symbols together. Further we assume that the changes made are relatively small and most changed symbols are related to this change. Thus to reduce this source of divergence, we want to keep related symbols together.

- Symbols can be created/deleted. This is relatively rare, and mostly covered by previous point.

- Symbols can be partitioned into different partition. This is main target of our partitioning strategy. We want to minimize moving symbols from one partition to another. But we do not care about moving individual symbols, only about whole partitions.

As secondary goal we want partitioning where partitions have similar sizes, so that LTRANS phase can be reasonably parallelized. The sizes are counted from amount of instructions which are only estimated.

## 3.1 Existing strategies

There already exist several partitioning strategies. None of them are ideal for our use case.

- Partitioning strategy `balanced` is current default. It tries to create $N$ equally sized partitions. This leads to relatively small binaries and good parallelizable compile times. We want to achieve similar results but also reduce chances of partitions changing as outlined at the start of chapter.

- Partitioning strategies `1to1` and `max` are intended for GCC development, where single partition corresponds to single source file or single symbol respectively. Such small partitions would be beneficial for caching. However our main reason for incremental compilation is to work on identical final binary which will be deployed. Thus the partitioning strategy must be also suitable for compiling from scratch and then deploying.

  Both `max` and to lesser extent `1to1` result in a lot of small partitions. For small partitions WPA must stream more data that would otherwise be shared inside a partition. These duplicates then often persist to final binary, significantly increasing its size. It also inhibits some optimizations, such as inter–procedural registry allocation, which happen locally within a single partition.

Interestingly we found that in our quick single measurement, while `1to1` was slower in real time, total time was faster than `balanced`. So the longer real time was only because some individual compiled files contain almost 10 % of all program instructions. Exact reason why `balanced` was slower is yet unclear, likely the tuned parameter of number of partitions is outdated. In any case, our partitioning strategy starts with results of `1to1` and then joins xor divides the files to equalize partition sizes. Thus if there is any fundamental benefit of `1to1` over `balanced`, our implementation is likely to get it as well.

- Partitioning strategies `one` and `none` which create single partition and compile everything directly in WPA respectively. Neither are useful for caching since there is only single partition (or none at all).

## 3.2 Design

Our partitioning strategy is implemented in 5th patch of A.1.

### 3.2.1 Files

We keep contents of a source file together, to keep related symbols together. This is also beneficial for partitioning symbols to the same partition. We always move whole files from one partition to another, which is less frequent than if individual symbols could change partition slowly one by one.

Most files are small enough so we can combine multiple files into one partition. If the files are too large we can split a file into multiple partitions. But for simplicity and reducing sources of divergence, we do not partition generally multiple files into multiple partition.

### 3.2.2 Partition set

*Partition set* is here used as a name of intermediate representation that contains groups of symbols and number of final partitions `n_partitions` that these symbols will be partitioned into. Groups of symbols are used to represent files from previous section and should be kept undivided as long as possible.

Partition sets do not influence each other and do not depend on global state. If partition set contains symbols with identical instruction sizes, and has identical `n_partitions`, distribution of symbols into resulting partitions will be identical.

Partition set with `n_partitions=1` is equivalent to final partition with contained symbols.

Only allowed operation partition set is to split it into multiple partition sets. These new partition sets must disjointly contain all original symbols and their sum of `n_partitions` should be the same as original.

All partition sets also contain additional metadata, such as whether symbol groups are already split into individual symbols.

Target size is simply total instruction count of whole partition set divided by `n_partitions`. In ideal situation this would be the size of all partitions resulting from this partition set.

### 3.2.3 Partitioning phases

Partitioning phases are procedures that split a partition set into multiple partition sets. The phases are intentionally depending on as little information as possible to reduce divergence.

Each symbol has a defined order in which it appeared in source files or when it was created during compilation. Some symbols are sensitive to this order and require to be in the same relative order in the final binary. But most symbols are not sensitive to change of order, so we can change it most of the time. However by trying to conserve order of symbols in most phases, we also reduce sources of divergence. Without conserving order, we can pick and choose any symbol groups to be in new partition sets. With conserving order, we can only choose single order which divides all symbol groups into two sets.

At the start of development there were attempts to disregard symbol order and to join symbol groups which are most connected by for example function calls between them. Such connections would mean that the symbols are related and by joining such related groups we concentrate changes to less partitions. This reduction was observed in average test cases, however divergence of partitioning symbols to different partitions is much more common. Thus this was abandoned and simpler phases which conserve order are now used.

#### Distribution of n_partitions

During many phases we first separate symbols into partition sets and only then we assign `n_partitions`. In such cases we try to achieve fairness in distributing `n_partitions` by following algorithm:

```
forall i:
 new[i].n_partitions = floor(old.n_partitions * new[i].size / old.size)
 new[i].n_partitions = max(new[i].n_partitions, 1)
while (sum over i: new[i].n_partitions) <= old.n_partitions:
 forall i:
  target_size[i] = new[i].size / new[i].n_partitions
  new_target_size[i] = new[i].size / (new[i].n_partitions + 1)
 j = index with max (target_size[j] + new_target_size[j])
 new[j].n_partitions += 1
```

Where `old` is the original partition set, `new` is list of all new partition sets the original is divided into. The algorithm assumes that amount of new partition sets is lower than total `n_partitions`.

In the first loop we add to each new partition the safe lower estimate of `n_partitions`. Each partition set must be partitioned into at least one partition, so we add this lower limit. In doing so, it is possible to overflow the total `n_partitions`. In these rare cases, there is also the reverse of the second loop, which removes `n_partitions` from partitions with smallest target size.

The second loop tries to fairly distribute rest of `n_partitions`. We try to maximize equivalent of:

```
(target_size[j]-old.target_size)-(old.target_size-new_target_size[j])
```

The basic assumption behind this is that in ideal case each new partition set would have target size equivalent to `old.target_size`. Thus we consider it to be positive when target size decreases to this value, and negative when it decreases further.

### Fixed split

Fixed split is the simplest phase intended to be the first. It is designed to not depend on any information that changes in common edit–compile cycle, so that divergence in partitioning is never global.

Only number of files is considered. Partition set is split into constant number of partition sets where all have the same (or off by one) number of symbol groups – files. Ordering of symbols is conserved. Partition sets' `n_partitions` are then distributed based on their sizes.

As an example in Figure 3.1 we split partition set with 42 files/ symbol groups and total `n_partitions`=128 into 4 partition sets with fixed split. The symbol groups are shown ordered, with width of each rectangle representing instruction size of given group. The longer lines show where the splits will be separating new partition sets. For each new partition set, files/symbol groups count and `n_partitions` are shown.

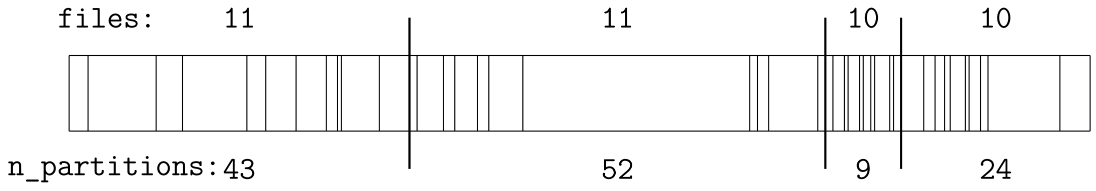

Figure 3.1: Fixed split

### Over target split

Over target split separates groups (files) that are larger than partition set's target size. Then `n_partitions` are distributed based on sizes of partition sets.

Then if a new partition set is large enough to be split further `n_partitions>1`, its single group is divided into groups where each group is a single symbol.

As example in Figure 3.2 we split partition set with total `n_partitions`=18. In total 3 symbol groups, shown with angled lines interior, were larger than the target size and are split of from the remaining groups. First two are large enough to be split into 4 and 3 `n_partitions` and are split into individual symbols. Third one is small with `n_partitions`=1 and is kept as is. Rest of groups are in one new partition set.

### Binary split

Binary split divides partition set into two partition sets with equal (or off by one) `n_partitions`. Ordering of symbols is conserved.

For even `n_partitions` symbol groups are split by finding the middle of all symbols by their size, and then snapping to the closest group boundary.

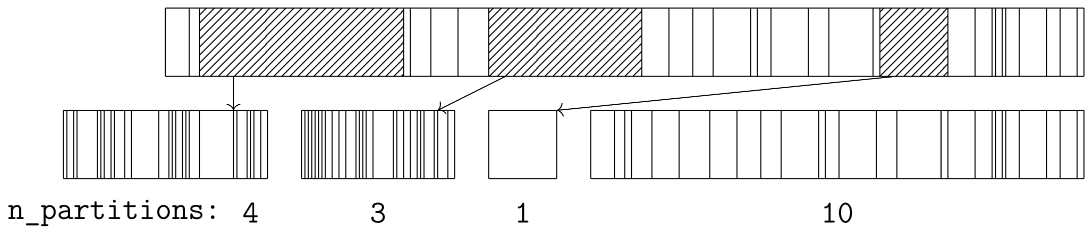

Figure 3.2: Fixed over target

For odd `n_partitions` instructions counts/distances are scaled to match nonequal `n_partitions` in resulting partition sets.

As an example in Figure 3.3 we split partition set with total `n_partitions`=3. The symbol groups are shown ordered, with width of each rectangle representing size/instruction count of given group. Only full lines correspond to symbol group boundaries. First we find the ideal split, showed by long dashed line, if the sizes were in perfect ratio 2:1. Then we find the closest symbol boundary, but now adjusted by opposite ratio 1:2. The reasoning is that if we move the split to the right, those symbols are added to left 2 partitions, and vise versa, and we care mainly about maximum size of partition. The long full line shows the closest found boundary, with short dashed line showing corresponding distance on the other side scaled by the ratio.

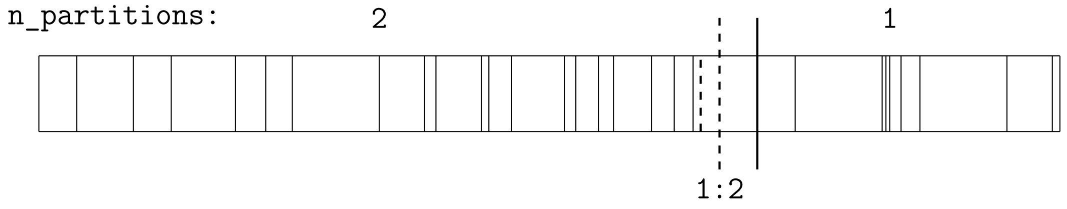

Figure 3.3: Binary split

### 3.2.4 Partitioner – phases composition

Partitioner decides which phases should be used for a given partition set. It is implemented using "dividing state machine". In each step we have single partition set with current state. The state describes which phase should be used to split the partition set. The state changes depending on the results of phase – and potentially differently for individual partition sets. Then each partition set with state is used individually in next step.

Default partitioner starts with Fixed divide, then continues with over target split and then binary division, which recursively returns to over target split, until `n_partitions=1`.

Basic partitioner is implemented to cover cases common for all possible partitioners. Thus implementation of previously mentioned default partitioner is:

```
virtual void
split_state (partition_set& p, uintptr_t state) {
 std::vector<partition_set> ps;
 switch (state) {
 case FIXED:
  if (p.n_partitions > 64 && p.sym_groups.size () >= 4) {
   ps = partition_fixed_split (p, 4);
   split_list (ps, OVER_TARGET);
   break;
  }
  /* FALLTHROUGH */
 case OVER_TARGET:
  ps = partition_over_target_split (p);
  if (!ps.empty ()) {
    split_list (ps, BINARY);
    break;
   }
  /* FALLTHROUGH */
 case BINARY:
  ps = partition_binary_split (p);
  split_list (ps, OVER_TARGET);
  break;
 }
}
```

Where `FIXED` must equal 0 to be the first state. Function `split_list` recursively calls `split_state` for all new partition sets in the list. But first it handles cases where the `n_partitions` is too small or too large to fit into limits, and if `n_partitions=1` it is finalized into partition.

# 4. Sources of divergence

Our cache will work best if as many as possible partitions do not change during edit–compile cycle. Previous chapter minimizes number of changed partitions by trying to concentrate changes into few partitions. However it would be better if those changes did not exist in the first place.

There are variables whose exact values don't influence quality of resulting binary. They might not influence the resulting binary at all or their influence is confined to unintentionally being a random seed. These values (or their influence) are stored in `ltrans.in` which on change make caching impossible, even if the `ltrans.out` would be always identical.

In this chapter we try to find as many such sources of divergence and try to find remedies.

## 4.1 Order of symbols

Each symbol is given `order` value which defines order in which the symbol appeared in source files. Most symbols are not sensitive to their order, but for some the order is important to conserve in final binary. The order is also given to symbols created during compilation, but it mainly useful for dumping symbol names to uniquely identify them.

This introduces a divergence, that if we introduce new symbol in the first file, every following symbols' order is shifted by one.

We care only about relative order of symbols. In LTRANS phase we need to know order of symbols in local partition and not about other partitions. Thus we can remap `order` of symbols in any partition to first `N` natural numbers, where `N` is number of symbols, while conserving their relative order. This way the local order is no longer influenced by outside symbols.

This problem is solved by patch number 6 of A.1.

There is a problem with symbols that were created during compilation intended to be inlined in LTRANS phase. They are sometimes created in different order and their order diverges. Since they will be inlined, their order does not matter and we can use order 0 for all such symbols. But it would be better to find and fix source of this divergence as well.

## 4.2 Order of inlining

During WPA phase there is an inlining pass. During inlining pass, inlining candidates are inserted into Fibonacci heap ordered by their badness. Badness is computed based on sizes of functions that should be inlined and that should inlined into. If both of their sizes are identical, badness will be identical as well. If two inlining candidates have identical badness their relative order of inlining is badly defined and will be dependent on exact state of Fibonacci heap, which is influenced by all other unrelated candidates.

To prevent this divergence, we added unique ID of edge, corresponding to the inlining candidate, as secondary comparison in Fibonacci heap. This way their relative order is always well defined.

This problem is solved by 10th patch of A.1.

## 4.3 Global Counters — Unique Identifiers

A lot of identifiers must be unique and by default this is achieved by using a global counter. Global counters allow local changes to leak to unrelated contexts.

Typically the identifiers are in format `name.counter` and we can generally fix this information leak by using `name.name_of_local_context.local_counter` format. We have to ensure that `name_of_local_context` is unique or that `local_counter` is shared among identically named local contexts.

### 4.3.1 tmp_variables

During compilation of initial compilation units additional variables are created which are differentiated by counter global to the compilation unit. Example can be a static variable inside of function.

```
void foo () { static int a; }
void bar () { static int a; }
```

The static variable persist to next function call, and thus must be elevated to the global scope of the file. Without renaming, the variables from `foo` and `bar` would collide.

In this case local context is a single function and global context are whole compilation unit. Although this counter is only global to a compilation unit, and a single compilation unit is partitioned together, this counter can leak to other partitions through inlining.

Originally the variable names are in format `name.counter`, or `a.0` and `a.1`. To reduce divergence we can use `name.function_name.local_counter`, or `a.foo.0` and `a.bar.0`. In case of variables local to a function, global uniqueness is not important and we can use `name.local_counter` as new identifier.

Partial fix is implemented in 9th patch of A.1.

### 4.3.2 lto_priv

In WPA after partitioning, we handle a case where multiple LGEN units are joined into one partition, while they both contained static variable with identical name. To prevent collision, we rename them into `name.lto_priv.counter` format.

In this case we can use partition where this collision happened as local context. Thus `name.lto_priv.partition_checksum.local_counter` as new identifier can be used to reduce divergence.

The 7th patch of A.1 for this divergence is in very experimental state, but it fulfills its role of removing divergence at least for our test cases.

## 4.4 Debug Information

Because of early debug, debug information is not directly contained in `ltrans.in`. It only contains symbol of the source file and offset identifying the debug information in that file. Unfortunately the symbol identifier is created as filename with hash of the source file to uniquely identify the file.

In principle we only need a stable unique identifier covered by previous section. We already have a format that is equivalent to `file_indentifier.counter`, however because the identifier is created in compile unit of the source file and has no knowledge about the outside world, it attaches unstable file hash to prevent filename collision. If we would know that there will be no filename collision, we could remove the hash and its divergence.

To remove the divergence from offset is more complicated. In principle we could use formats such as `filename.function_name.local_counter`. However the file identifier and offset is part of debug format DWARF [17].

## 4.5 Source file order

In partitioning we assumed that the source files have a constant order. However if the resulting object files are contained in an archive file, they are loaded on demand. So if a symbol is used earlier the whole file will be ordered earlier.

## 4.6 Random seed in section names

Object files are divided into multiple sections that contain symbols. Section names were named in format `symbol.random_seed`. This is useful for incremental linking. However it is mutually exclusive with LTO, so we can simply use 0 instead of `random_seed`.

Solved by patch number 2 of A.1.

## 4.7 Flags in object files

Object files, and `ltrans.in` specifically, contain list of all flags that were used for compilation. These are useful for caching, because if we use different flags, the basic assumption that identical `ltrans.in` files result in identical `ltrans.out` would be broken.

However not all flags influence the compilation. Many such flags were excluded already, but there is an unexcluded flag which specifies a filename into which WPA prints list of `ltrans.in` filenames. This filename must change to allow parallelization, but does not change anything during LTRANS phase.

Removed by patch number 1 of A.1.

# 5. Usage guide

Some parts (filelocks) of the implementation are for now only implemented for POSIX systems, thus Linux is recommended to test it.

## 5.1 Building modified GCC

To build modified GCC, we start specifically with GCC's commit `423d34f61c4` from 14th of March 2023. This is the last commit we tested and future commits may conflict with our patches, which means then won't be trivially appliable. We create the git directory `$GCC_DIR`. All directory paths are for simplicity absolute paths. We also create branch `ltrans_cache` to track our patches.

```
git clone git://gcc.gnu.org/git/gcc.git $GCC_DIR/
cd $GCC_DIR/
git checkout 423d34f61c4
git switch -c ltrans_cache
```

Then we apply all patches in attachment A.1 to our branch. The 8th patch is optional for debugging/measuring of our implementation, but we will assume it is applied.

```
git am *.patch
```

To build it, we create new directories `$BUILD` where GCC will be built and `$INSTALLED` where GCC will be installed. These directories must be outside of `$GCC_DIR`. Then we can build modified GCC:

```
mkdir $BUILD $INSTALLED
cd $BUILD
$GCC_DIR/configure --disable-bootstrap --prefix=$INSTALLED/ \
  --enable-languages=c,c++,lto --enable-checking=release
make -jN
make install
```

To use the modified GCC in existing build systems, we need to export `$CC` and `$CXX` variables specifying compilers to be used for C and C++ respectively. To make exporting easier, we setup sourced shell script `$SOURCE`, which we source with '`. $SOURCE`' in any shell instance used for compiling or setting up build systems:

```
#!/bin/sh
INSTALLED=Fill in the full path $INSTALLED directory
GCC_BIN=$INSTALLED/bin

export CC=$GCC_BIN/gcc
export CXX=$GCC_BIN/g++
```

## 5.2 Compiling with incremental LTO

To compile projects with our modified GCC, we need to export `$CC` and `$CXX` so that the build system of the project uses theses compilers. To enable LTO and our partitioning strategy we have to add compiler flags `-flto-partition=cache` and `-flto`. To enable LTO incremental compilation we only need to export `$GCC_LTRANS_CACHE` variable to point to existing directory where the cache will be located. Simple LTO incremental compilation example would be:

```
. $SOURCE
export GCC_LTRANS_CACHE=$(realpath cache/)
mkdir $GCC_LTRANS_CACHE
$CXX -O2 -flto -flto-partition=cache -c *.cc
$CXX -O2 -flto -flto-partition=cache *.o -o executable
```

## 5.3 Observing incremental compilation

With debug patch applied, during incremental compilation following information is print out:

- Instruction sizes of files and final partitions. These are useful to assess fairness of our partitioning strategy.
- In how many bytes each partition differs from first compilation. We use this to find which partition differs by only few bytes and thus are likely to contain fixable divergence. This information is reliable only for first recompilation.
- Which partitions were cached and which were recompiled. We use this to measure success of our incremental compilation.

For too small programs incremental compilation is not useful, because they are partitioned into too few partitions. In such cases it is possible to reduce the minimal partition size with compilation flag `--param lto-min-partition=10000` which defaults to 10000. Successful caching was observed with programs with as little as 7 partitions.

# 6. Results

To measure results we build the compiler and modify our test projects to be compiled with incremental LTO as described in previous chapter.

To test our implementation we measure our success by number of partitions that were cached/recompiled for a given edit–compile cycle. We want to emulate edit–compile cycles with representative real world examples. For a given program, we start with a benchmark base corresponding to master branch in its git repository. Then we pseudorandomly select individual git commits which will be reverted/applied to the benchmark base. These commits will be our test cases.

## 6.1 cc1 – debug(less)

We used GCC's C compiler `cc1` as our main target with many test cases to find divergences in previous sections. The benchmark base we use is commit `f8cb07a7a44` from 14th January 2023. The reverted/applied commits are from around similar time.

We measured compilation both with debug symbols and without them. The results are in Figure 6.1 where we compiled with 128 partitions, the names correspond to hashes of reverted/applied commits. In case of incremental compilation without debug symbols we have consistent high fraction of cache hits. On average only roughly 1/6 of partitions is recompiled. In the case with debug symbols the consistency is much worse, but the average number of cache hits is still worthwhile, on average only roughly 1/3 of partitions is recompiled.

Important test case is `b1f30bf42d8` which has the most recompilations. This is because this test case is an example of diverging source files order mentioned in 4.5. So this case is likely to improve. The other interesting test case is `10bd26d6efe` because the case with debug symbols has more cache hits than debugless case. It is yet unclear why.

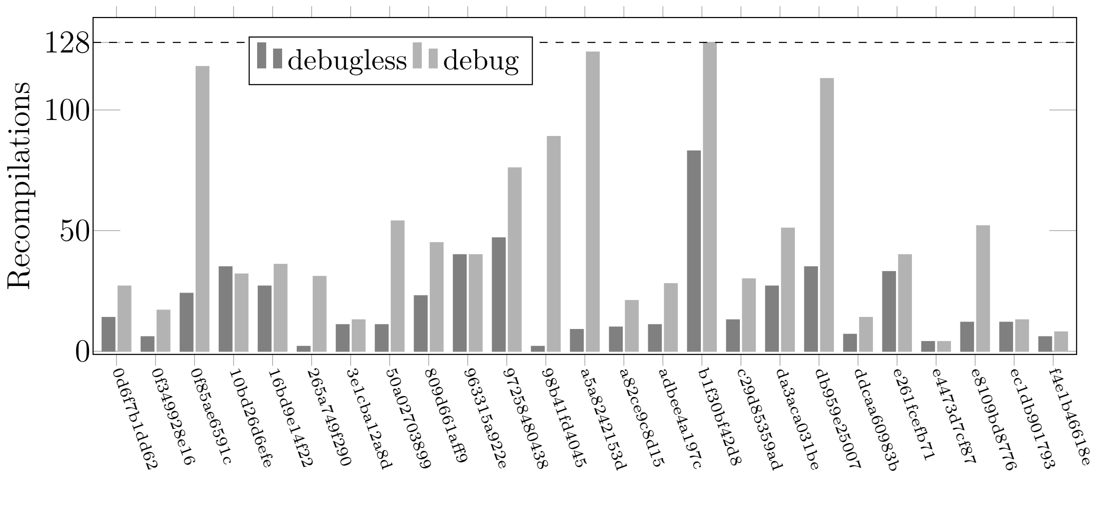

Figure 6.1: Edit–compile cycles of cc1

The exact number of recompilations are in attachment A.3.

## 6.2 Other programs

After finding divergences we also tested other programs to make sure the solution works in general conditions. We only try few cases to get very rough estimate of recompilations.

### 6.2.1 Clang

Clang[1] is C/C++ frontend for LLVM. We have chosen it as another large program which has 3 times more instructions than already tested GCC's cc1. We measure it the same way as cc1, we start with benchmark base being commit `7adf3849e40e` from 10th of May 2023.

Results are in Figure 6.2 and the values are in A.4. The results are similar to `cc1`, but the `d1d35f04c6cb` and `6db007a0654e` require more recompiling than we expected from debugless. It is likely because they contain header files, which cause multiple LGENs to recompile.

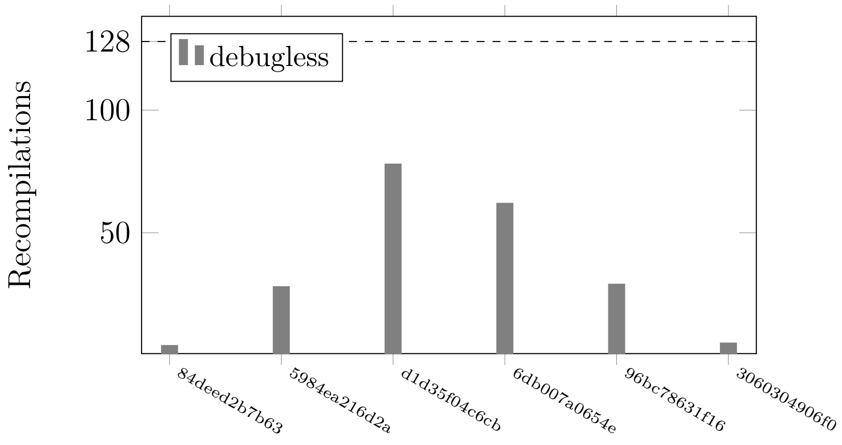

Figure 6.2: Edit–compile cycles of clang

### 6.2.2 Athena++

As an example of a small program we used Athena++[2] which is partitioned into 15 partitions. Since it is code for astrophysical simulations, most changes happen in the problem description source file. So we tried to modify problem description file (specifically *disk*, configured in spherical coordinates) in several ways:

- Modifying contents of functions consistently leads to recompiling 3–5 of 15 partitions.
- Removing one of the two virtual functions leads to recompiling either 2 or 9 of 15 partitions.
- Removing everything leads to recompiling 11 of 15 partitions.

[1]https://clang.llvm.org/
[2]https://www.athena-astro.app/

## 6.3 Cache vs balanced partitioning

With our new `cache` partitioning strategy there may be concern that the partition sizes will be less uniform than previous `balanced` strategy which could negatively affect parallelization. So we compare partition sizes of those two partitioning for benchmark base used in case of `cc1` specifically commit `f8cb07a7a44` in Figure 6.3, data are in A.5. Partitions are sorted by their instruction sizes to see their size distribution. Average size of partitions is 65637.

We can see that the size distributions are similar with massive difference in the largest partitions, where the old `balanced` strategy fails. Sizes of `balanced` are from 40505 to 162594, while sizes of `cache` are from 43987 to 87654. So if anything we would expect better parallelization with `cache`.

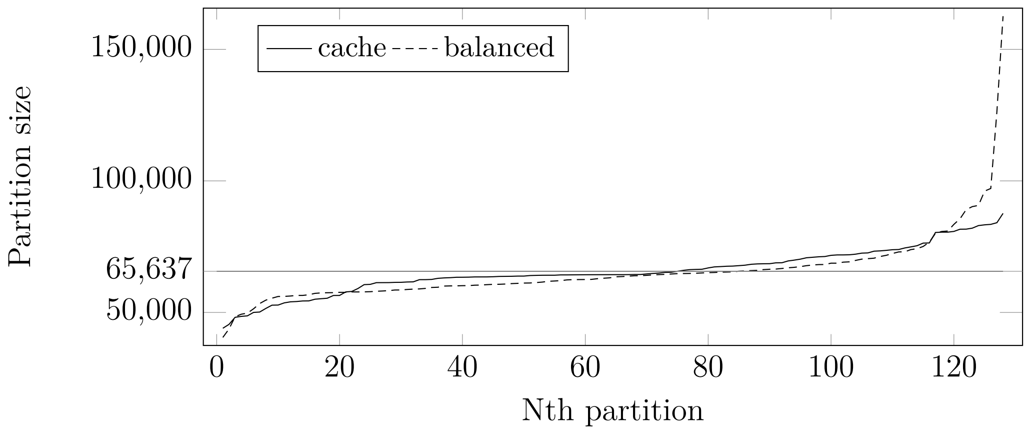

Figure 6.3: Partition sizes when compiling cc1

# Conclusion

GCC's WHOPR model of LTO (see Section 1.5) consists of two link–time phases WPA and LTRANS, where WPA analyzes the whole program and parallelized LTRANS does majority of work. This is great scalability improvement over naive single global phase. Natural extension of this idea is to LTO incrementally and reuse results of LTRANS from previous compilations to reduce edit–compile cycle times.

Incremental Link–Time Optimization, as implemented here, significantly reduces work required during local transformation phase and thus reduces edit–compile compile times, most importantly on machines with low core count. It works well with both large (GCC's cc1, Clang) or small (Athena++) programs.

On average roughly 1/6 of partitions need to be recompiled without debug information. And there are still several divergences that can be fixed to make incremental compilation more successful. The needed recompilations are less consistent and greatly increased on average to 1/3 of partitions when debug information is enabled. This is not a fundamentally unsolvable problem, however it requires changes to how debug information is indexed, which is part of DWARF debug format.

Our results show that incremental LTO in GCC is practically usable. It is planned to make these changes in some form part of official GCC. However first we need to fix problems with debug information caused by early debug. This was an unexpected problem and needs to be discussed with maintainers of debug info in GCC.

A pleasant side effect of incremental LTO is a motivation to improve LTO infrastructure. First, we remove divergences which are undesirable even without incremental compilation. Small changes to part of a program should not influence other parts without a good reason. Second, the global WPA becomes the bottleneck, which motivates its improvement. Which would be otherwise noticeable only to highly parallelized systems.

Still, other avenues to LTO exist. LLVM contains substantially different implementation of scalable LTO called ThinLTO (see Section 1.6.2). Benefit of ThinLTO is almost full elimination of WPA at cost of less optimizations working over the whole program. It is possible to implement hybrid between ThinLTO and GCC's WHOPR, but it is unclear how many users are not satisfied with `-O2` but would compromise build time at cost of optimization opportunities.

# Bibliography

[05] *Link-Time Optimization in GCC: Requirements and High-Level Design.* 2005. URL: `https://gcc.gnu.org/projects/lto/lto.pdf` (visited on 05/07/2023).

[17] *DWARF Debugging Information Format Version 5.* DWARF Debugging Information Format Committee. 2017. URL: `https://dwarfstd.org/dwarf5std.html` (visited on 05/07/2023).

[Bie15] Richard Biener. *GCC wiki page Early Generation of Debug Information.* 2015. URL: `https://gcc.gnu.org/wiki/early-debug` (visited on 05/07/2023).

[Bri+07] Preston Briggs et al. *WHOPR — Fast and Scalable Whole Program Optimizations in GCC.* 2007. URL: `https://gcc.gnu.org/projects/lto/whopr.pdf` (visited on 05/07/2023).

[GH10] Taras Glek and Jan Hubička. “Optimizing real world applications with GCC link time optimization”. In: *Proceedings of the 2010 GCC Developers' Summit.* 2010, pp. 25–46.

[Hub14] Jan Hubička. *Devirtualization in C++.* Blog post. 2014. URL: `http://hubicka.blogspot.com/2014/01/devirtualization-in-c-part-1.html` (visited on 05/07/2023).

[Hub21] Jan Hubička. *New IPA-modref pass for GCC.* GNU Tools Cauldron. 2021. URL: `https://lpc.events/event/11/contributions/1016/attachments/897/1713/modref.pdf` (visited on 05/07/2023).

[JAL17] Teresa Johnson, Mehdi Amini, and Xinliang David Li. “ThinLTO: scalable and incremental LTO”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE. 2017, pp. 111–121.

[Jam09] Martin Jambor. “Interprocedural optimizations of function parameters.” In: *Proceedings of the 2009 GCC Developers' Summit.* 2009, pp. 57–63.

[LAH10] David Xinliang Li, Raksit Ashok, and Robert 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.

[Liš14] Martin Liška. “Optimizing large applications”. Master Thesis. Charles University, 2014.

[LN05] Razya Ladelsky and Mircea Namolaru. “Interprocedural constant propagation and method versioning in GCC”. In: *Proceedings of the 2005 GCC Developers' Summit.* 2005, pp. 133–144.

[Muc+97] Steven Muchnick et al. *Advanced compiler design implementation.* Morgan kaufmann, 1997.

[Tho90] Ken Thompson. “Plan 9 C compilers”. In: *Proceedings of the Summer 1990 UKUUG Conference.* 1990, pp. 41–51.

# A. Attachments

## A.1 Patches

GCC's source code modifications divided into 10 patches. It is in electronic form only.

## A.2 LTO times

With 24 threads, compiling `cc1plus` the total compile time of WPA and LTRANS was 52.620s real time, 12m36.361s use time. Measured compile time of WPA and LTRANS in total, and measured individual passes of WPA. Following is list of times spent in individual WPA passes, created by `-ftime-report`.

```
Time variable                                   usr           sys          wall           GGC
 phase setup                        :   0.00 (  0%)   0.00 (  0%)   0.00 (  0%)  2583k (  0%)
 phase opt and generate             :  11.07 ( 84%)   0.15 ( 20%)  11.23 ( 79%)   770M ( 46%)
 phase stream in                    :   1.79 ( 14%)   0.19 ( 26%)   1.98 ( 14%)   887M ( 53%)
 phase stream out                   :   0.06 (  0%)   0.39 ( 53%)   0.75 (  5%)  4096  (  0%)
 phase finalize                     :   0.30 (  2%)   0.01 (  1%)   0.32 (  2%)     0  (  0%)
 garbage collection                 :   0.32 (  2%)   0.02 (  3%)   0.35 (  2%)     0  (  0%)
 callgraph optimization             :   0.17 (  1%)   0.00 (  0%)   0.16 (  1%)  7560  (  0%)
 ipa ODR types                      :   0.12 (  1%)   0.00 (  0%)   0.16 (  1%)     0  (  0%)
 ipa function summary               :   0.29 (  2%)   0.05 (  7%)   0.32 (  2%)   186M ( 11%)
 ipa dead code removal              :   0.73 (  6%)   0.01 (  1%)   0.75 (  5%)     0  (  0%)
 ipa devirtualization               :   0.01 (  0%)   0.00 (  0%)   0.01 (  0%)   313k (  0%)
 ipa cp                             :   0.39 (  3%)   0.01 (  1%)   0.40 (  3%)    34M (  2%)
 ipa inlining heuristics            :   5.54 ( 42%)   0.05 (  7%)   5.58 ( 39%)   560M ( 34%)
 ipa comdats                        :   0.13 (  1%)   0.00 (  0%)   0.13 (  1%)     0  (  0%)
 lto stream decompression           :   0.24 (  2%)   0.01 (  1%)   0.25 (  2%)     0  (  0%)
 ipa lto gimple in                  :   0.10 (  1%)   0.03 (  4%)   0.15 (  1%)    77M (  5%)
 ipa lto decl in                    :   0.76 (  6%)   0.06 (  8%)   0.81 (  6%)   524M ( 32%)
 ipa lto constructors in            :   0.01 (  0%)   0.00 (  0%)   0.00 (  0%)  3442k (  0%)
 ipa lto cgraph I/O                 :   0.06 (  0%)   0.03 (  4%)   0.08 (  1%)   109M (  7%)
 ipa lto decl merge                 :   0.08 (  1%)   0.00 (  0%)   0.08 (  1%)  2564k (  0%)
 ipa lto cgraph merge               :   0.10 (  1%)   0.00 (  0%)   0.10 (  1%)     0  (  0%)
 whopr wpa                          :   0.17 (  1%)   0.01 (  1%)   0.21 (  1%)  8192  (  0%)
 whopr wpa I/O                      :   0.05 (  0%)   0.39 ( 53%)   0.73 (  5%)  4096  (  0%)
 whopr partitioning                 :   0.74 (  6%)   0.00 (  0%)   0.73 (  5%)   853k (  0%)
 ipa reference                      :   0.67 (  5%)   0.00 (  0%)   0.67 (  5%)     0  (  0%)
 ipa profile                        :   0.05 (  0%)   0.00 (  0%)   0.04 (  0%)     0  (  0%)
 ipa pure const                     :   0.85 (  6%)   0.01 (  1%)   0.86 (  6%)     0  (  0%)
 ipa icf                            :   0.59 (  4%)   0.01 (  1%)   0.60 (  4%)    26M (  2%)
 ipa SRA                            :   0.30 (  2%)   0.02 (  3%)   0.31 (  2%)    64M (  4%)
 ipa free inline summary            :   0.14 (  1%)   0.00 (  0%)   0.14 (  1%)     0  (  0%)
 ipa modref                         :   0.57 (  4%)   0.01 (  1%)   0.58 (  4%)    47M (  3%)
 tree SSA incremental               :   0.00 (  0%)   0.00 (  0%)   0.01 (  0%)  1234k (  0%)
 tree operand scan                  :   0.02 (  0%)   0.02 (  3%)   0.03 (  0%)    17M (  1%)
 tree modref                        :   0.00 (  0%)   0.00 (  0%)   0.00 (  0%)  1025k (  0%)
 dominance computation              :   0.01 (  0%)   0.00 (  0%)   0.01 (  0%)     0  (  0%)
 varconst                           :   0.00 (  0%)   0.00 (  0%)   0.02 (  0%)     0  (  0%)
 TOTAL                              :  13.22          0.74         14.28         1660M
```

## A.3 GCC's cc1 recompilation counts

Recompilation counts for GCC's cc1 both with and without debug info. Names of commits contain either part of commit message, or names of files they modified.

```
debugless debug commit
35/128 113/128 b_rev_db959e25007_ipa
83/128 128/128 b_rev_b1f30bf42d8_Fix_wrong_code_issues_with_ipa-sra
 2/128  89/128 b_rev_98b41fd4045_c_c++_Allow_ignoring_-Winit-self_through_pragmas
24/128 118/128 b_rev_0f85ae6591c_c_ICE_with_nullptr_as_case_expression
11/128  28/128 b_rev_adbee4a197c_tree-optimization
11/128  13/128 b_rev_3e1cba12a8d_tree-sra_cc
12/128  13/128 b_rev_ec1db901793_i386_lujiazui_md
 9/128 124/128 b_rev_a5a8242153d_cfgexpand_cc
11/128  54/128 b_rev_50a02703899_varpool_cc
 7/128  14/128 b_rev_ddcaa60983b_loop-invariant_cc
27/128  36/128 b_apply_16bd9e14f22_tree-ssa-loop-niter_cc
 2/128  31/128 b_apply_265a749f290_lra-constraints_cc
13/128  30/128 b_apply_c29d85359ad_tree-ssa-sccvn_cc
```

```
10/128  21/128 b_apply_a82ce9c8d15_opts_cc
27/128  51/128 b_apply_da3aca031be_ipa-utils_cc
40/128  40/128 b_apply_963315a922e_i386_i386-expand_cc
47/128  76/128 b_apply_97258480438_dominance_cc
 4/128   4/128 b_apply_e4473d7cf87_ree_cc
 6/128  17/128 b_apply_0f349928e16_tree-nested_cc
 6/128   8/128 b_apply_f4e1b46618e_rtl-ssa_accesses_cc
12/128  52/128 b_apply_e8109bd8776_ipa-{sra,param-manipulation}_cc
35/128  32/128 b_apply_10bd26d6efe_range-op-float_cc
33/128  40/128 b_apply_e261fcefb71_value-range_cc
14/128  27/128 b_apply_0d6f7b1dd62_analyzer_h
23/128  45/128 b_apply_809d661aff9_range-op_cc_h
```

## A.4 Clang recompilation counts

Recompilation counts for clang, only without debug info.

```
debugless commit
 4/128 84deed2b7b63
28/128 5984ea216d2a
78/128 d1d35f04c6cb
62/128 6db007a0654e
29/128 96bc78631f16
 5/128 3060304906f0
```

## A.5 Partition sizes

Sorted partition sizes when compiling cc1, with new *cache* and old *balanced* partitioning strategy. Sizes of partitions are expected instruction counts as computed by inline heuristics.
Cache:

```
43987 45460 48150 48503 48705 50015 50156 51525
52766 52811 53641 54032 54150 54364 54392 55028
55220 55404 56450 56465 57797 57991 59106 60569
60677 61319 61330 61341 61435 61451 61575 61630
62458 62467 62550 62963 63126 63213 63371 63382
63416 63500 63504 63510 63571 63691 63718 63743
63815 63818 64026 64095 64137 64148 64148 64272
64272 64291 64308 64327 64336 64342 64347 64355
64359 64360 64372 64426 64429 64639 64823 64967
65228 65272 65650 66010 66264 66356 66416 66921
67307 67422 67558 67618 67783 67961 68302 68447
68504 68527 68852 68881 69567 69884 70252 70781
70971 71152 71255 71706 71862 71868 71918 72149
72542 72624 73287 73400 73600 73839 73914 74504
74921 75358 76354 76413 80392 80516 80519 80795
81634 81660 82040 83016 83298 83474 84107 87654
```

Balanced:

```
40505 43692 48371 49325 49739 51378 53386 54675
55421 56068 56102 56244 56469 56475 56720 57257
57430 57489 57522 57640 57641 57797 57798 57814
57889 58047 58154 58286 58528 58542 58735 58868
58942 59143 59496 59580 60028 60087 60104 60175
60196 60357 60409 60561 60684 60704 60800 60959
61057 61166 61198 61353 61541 61838 61954 62084
62369 62463 62496 62518 62565 62835 63011 63163
63544 63642 63783 63940 63973 64105 64269 64388
64472 64615 64749 64791 64823 64901 65008 65128
65190 65311 65359 65426 65678 65793 65904 66007
66277 66415 66482 66705 66970 67147 67408 67776
68031 68187 68218 68723 68724 69118 69232 69783
70258 70444 70621 71261 71719 72396 72999 73051
73885 74241 75082 76716 80057 80764 80948 83474
85613 89077 90228 90681 96114 97119 126365 162594
```