ThinLTO: Scalable and Incremental LTO

Teresa Johnson
Google, USA
tejohnson@google.com

Mehdi Amini
Apple, USA
mehdi.amini@apple.com

Xinliang David Li
Google, USA
davidxl@google.com

Abstract

Cross-Module Optimization (CMO) is an effective means for improving runtime performance, by extending the scope of optimizations across source module boundaries. Two CMO approaches are Link-Time Optimization (LTO) and Lightweight Inter-Procedural Optimization (LIPO). However, each of these solutions has limitations that prevent it from being enabled by default. ThinLTO is a new approach that attempts to address these limitations, with a goal of being enabled more broadly. ThinLTO aims to be as scalable as a regular non-LTO build, enabling CMO on large applications and machines without large memory configurations, while also integrating well with distributed and incremental build systems. This is achieved through fast purely summary-based Whole-Program Analysis (WPA), the only serial step, without reading or writing the program's Intermediate Representation (IR). Instead, CMO is applied during fully parallel optimization backends.

This paper describes the motivation behind ThinLTO, its overall design, and current implementation in LLVM. Results from SPEC cpu2006 benchmarks and several large real-world applications illustrate that ThinLTO can scale as well as a non-LTO build while enabling most of the CMO performed with a full LTO build.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; D.3.4 [*Processors*]: Incremental compilers; D.3.4 [*Processors*]: Optimization

Keywords Inter-procedural, Cross-module, Optimization, Link-Time Optimization

1. Introduction

Compilers normally operate on a single translation unit at a time. Each translation unit, which includes a single source file and its expanded headers, is compiled into a single native object file, and ultimately the linker combines multiple object files into a resulting binary or library. Optimizations are optionally applied within each translation unit (or module) during the compilation, at both the function scope and the module scope. The latter is referred to as Inter-Procedural Optimization (IPO). Function inlining [7][6] is a key IPO. However, normally a callee can only be inlined into its caller if they are defined in the same module.

In order to achieve larger benefits from IPO, the compilation scope can be increased to include multiple modules, thus enabling CMO. When the scope includes all modules being linked into an executable, WPA enables more aggressive optimizations that rely on visibility into all symbol definitions and uses. For example, Hall's dissertation [11] describes a mechanism for constructing a full call graph of the program, on which IPO decisions are made, with the results applied on the program representation before continuing with compilation. However, this broader optimization scope comes at both a memory and build time cost, limiting its scalability. The memory footprint is impacted by the larger in-memory data required to hold multiple modules' IR and analyses data. The build time cost is both due to the reduced available parallelism in the build process, as the modules are no longer compiled independently, and also because some algorithms scale non-linearly with the increased scope. Different approaches to CMO vary in how they combine multiple modules for analyses and optimizations, sometimes using summaries and partitioning to limit the memory and runtime cost.

One common mechanism for enabling CMO is through the use of LTO. While some implementations rely on binary decompilation [19, 20, 27], most production compiler implementations currently emit IR instead of object files. These object files are then fed to the linker, which invokes the compiler to perform the Inter-Procedural Analysis (IPA) and subsequent IPO, followed by other optimizations, and finally the code generation of the native executable. However, the CMO is usually performed across all modules in a serial step. In some implementations, though, the function level optimizations and code generation are subsequently performed in parallel.

The LLVM [23] [14] compiler implements monolithic LTO, which does all IPO compilation in serial. All modules are combined into a single monolithic module on which IPO is applied. However, there is support for parallel code generation, where the combined module is split into chunks after optimization. The first LTO implementation in the HP-UX compiler [3] was also operating on multiple files as a monolithic unit. It includes an advanced mechanism to limit the memory usage by loading only the IR needed for a transformation and reserializing to disk in between. While allowing

978-1-5090-4931-8/17$15.00

large applications to build on a workstation, it does not perform very well and is bound by the intensive I/O operations. The SYZYGY [18] high-level optimizer replaces it in the HP-UX Itanium compiler, making summary based inlining and other IPO decisions. However, inlining still occurs in serial, and the implementation requires multiple rewrites of the IR before parallelizing the back-end compilation. Similarly, the Open64 [2] (and related Open Research Compiler [13]) IPA implementation makes summary based decisions, but also rewrites the IR before launching parallel backends. In the GCC [8] compiler's LTO implementation (WHOPR) [9], IPA and inlining decisions are made in serial, operating on summaries built after reading the IR. The actual inlining transformations are done with some level of parallelization after dividing the call graph into multiple partitions. The implementation requires additional extensive I/O in the serial step to clone function bodies across partition boundaries.

All of these implementations require a serial IPA/IPO step that is both memory intensive and slow, as will be shown in Section 8.2, limiting usability on either smaller workstations or large applications. This is particularly true with debug information, which requires significantly more time and memory to represent and link. Additionally, fast incremental rebuilds are not possible in this model, since any change requires performing this expensive IPA/IPO step again. The Microsoft Visual Studio 2015 release includes support for incremental LTO [17], however, the very limited information available suggests that this is done by updating the binary with new versions of functions affected by the new IPA, and that the results are not the same as with a clean rebuild.

LIPO [15][16], which is implemented on GCC's Google branch, was designed to overcome the need for a serial linker plugin step. LIPO is tightly coupled with Feedback-Directed Optimization (FDO). It leverages the training run step of FDO to perform full program analyses at runtime (dynamic IPA). The implementation does a coarse-grain inlining analysis on the dynamic call graph built from call profiles at the end of the profile training run. That analysis identifies for each module a set of auxiliary modules containing its hot callees. In the profile-use build, each primary module is built independently, but is extended to include all of its auxiliary modules during parsing. LIPO scales much better than LTO, and works naturally with distributed build systems. However, it has several limitations, including its requirement for profiling, and a coarse-grain module grouping which leads to memory overhead. It also requires extensive changes in the language frontend and does not support cross language CMO since auxiliary modules are included in the frontend.

As described above, both LTO and LIPO have their own limitations and are not suitable for out of the box/general use. On the other hand, ThinLTO is designed to scale like a non-LTO build, enabling CMO on machines without large memory configurations, while retaining most of the performance achievement of LTO. It is also designed to integrate well with distributed build systems, and allows fast and stable incremental builds.

The ThinLTO serial phase in the linker plugin is designed to be thin and fast. ThinLTO exclusively utilizes compact summaries of each module for global analyses in the serial link step, without loading or writing any program IR. IPO transformations are performed later when the modules are optimized in fully parallel backends. A novel function importing transformation, made possible by the summary index built during the serial step, enables the inlining benefits provided by CMO.

While LIPO makes module grouping decisions at profile training runtime, ThinLTO only makes decisions at link time, with optional use of profile feedback. Unlike the coarse-grained LIPO module groups or WHOPR partitions, during the ThinLTO parallel backend phase every module imports just those functions useful for CMO. This finer-grained grouping minimizes the memory overhead, and maximizes the available parallelism.

We evaluate the performance of ThinLTO on the SPEC benchmark suite and show how it obtains almost all of the gain achieved through more expensive traditional LTO. Using three large real-world applications, we also demonstrate how ThinLTO scales much better than the LTO implementations in GCC and LLVM.

The rest of the paper is organized as follows. Section 2 describes the overall design of ThinLTO. Section 3 describes incremental build support. Section 4 describes the function importing transformation that enables cross-module inlining and indirect call promotion, and Section 5 describes other currently implemented summary-based Whole-Program Optimization (WPO). Section 6 describes distributed build system integration. Section 7 provides some details on the LLVM implementation, and experimental results are presented and discussed in Section 8. Finally, Sections 9 and 10 describe future directions and conclusions, respectively.

2. ThinLTO Design

The ThinLTO process is divided into 3 phases, illustrated in Figure 1:

1. Compile: Generate IR as with LTO mode, but extended with module summaries.
2. Thin link: Thin linker plugin layer to combine summaries and perform global analyses.
3. ThinLTO backend: Parallel backends with summary-based importing and optimizations.

These phases are detailed in the following subsections.

2.1 Compile Phase: Summary Emission

The first (compile) phase is similar to a traditional LTO compile step. The frontend is invoked to translate each input source file into an intermediate file containing IR, after some early optimizations (mostly for size reduction). However,

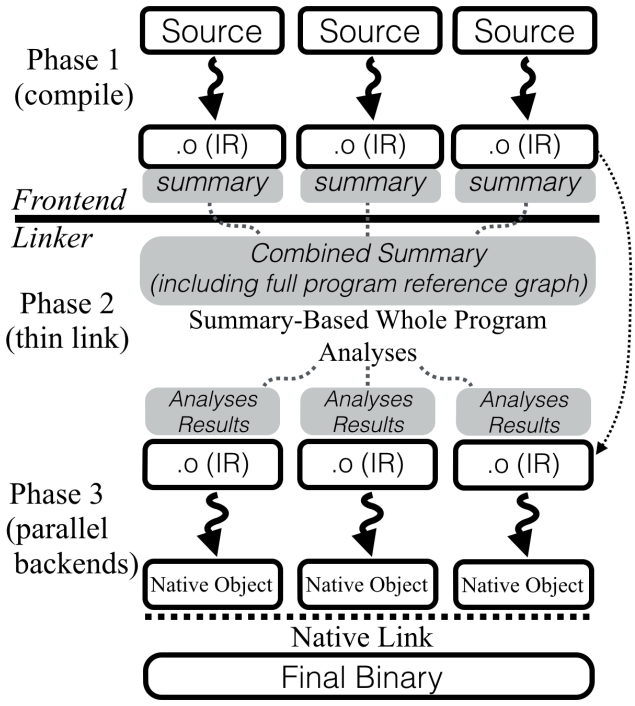


Figure 1: ThinLTO Overview

with ThinLTO an additional summary section is included in each file. In LLVM this conversion to IR and summary generation are performed in the middle end, so the frontend only needs to pass through the options. As a result, ThinLTO supports cross-language CMO, since it operates exclusively on IR.

This summary is the cornerstone of the ThinLTO design. It is emitted for each module into the object file containing its IR. These summary sections are designed so that they can be separately loaded without any expensive construction. Each global variable and function has an entry in the module summary. An entry contains metadata to drive the thin link global analyses. Currently, a function entry contains the linkage type, the number of instructions, optional Profile-Guided Optimization (PGO) [4] information, and some flags. The format is intended to be extended as needed. Additionally, every reference (address taken, load/store) and call (direct call targets, or indirect call targets discovered in value profile information) are recorded. Calls are optionally decorated with the PGO *hotness*, later used by global analyses (see Section 7.2).

2.2 Thin Link Phase

The second (thin link) phase is the only serial step in the ThinLTO scheme. Even though it is not a requirement, it would usually be implemented as a linker plugin to insert transparently in existing build systems. Also some analyses are more accurate when this phase is fed with extra information from the linker. This phase starts by reading just the summary sections from the IR files, and simply aggregating them into a single combined summary index. The reference and call edges recorded in each module's summary together model an accurate reference and call graph for the program. The IR itself is not parsed.

A key aspect of the ThinLTO model is that CMO must be split into two parts:

1. Analysis: This part is performed in the thin link serial phase and operates over the reference graph. To keep the thin link as fast as possible, only the summaries are available for the analysis. This avoids parsing and loading any IR in memory.
2. Transformation: This is performed during the parallel backends phase. It will use the result of the WPA to apply transformations on the IR.

This split model is key for ThinLTO high-scalability. An important example of such an optimization is the *function importing* transformation, which will be detailed in Section 4. Section 5 describes the implementation of additional global analyses and optimizations.

Additionally, the combined index is augmented to contain a module path symbol table, which identifies for each symbol the path to each IR file that defines it. In cases where a symbol with weak linkage or in a COMDAT [12] section has definitions in multiple modules, multiple summary entries may correspond to the same identifier. As described in Section 5.2, the linker can help identify *prevailing* copies of these symbols. Finally, to disambiguate local symbols from other local or global symbols with the same name, each local symbol is given a unique but deterministic global identifier in the index, formed by appending its defining module's path to the original name.

2.3 ThinLTO Backends

The third (ThinLTO backend) phase performs the backend compilation from IR to native code for each module in parallel, using the results of the earlier summary based analyses computed during the thin link phase. These results are used to perform the actual transformations independently for each module.

By default, linkers are setup to launch the ThinLTO threads in parallel from the linker process via a thread pool. So the distinction between the second and third phases is transparent to the user. Section 6 will show how a distributed build system would instead execute the backends independently on different remote build machines.

3. Incremental Builds

During software development, engineers rely on the build system's ability to re-compile the minimum amount of files. The widely used *make* uses explicit dependency tracking and timestamp comparisons to achieve this.

However, even modifying a single source file will always re-trigger the link step to produce the final binary. Unfortunately, with monolithic LTO, this final link step usually

dominates the total build time, annihilating the usually fast incremental build and limiting the use of LTO in practice.

ThinLTO is designed to integrate with incremental build systems. To this end, the process includes the following:

1. At the end of the first phase, while emitting the IR file, the content is hashed and the result is appended to the file. Section 7.1 details the serialization of the hash.
2. During the third phase (ThinLTO backend), the input IR file hash is combined with the analyses results from the second phase (thin link) to produce a new hash. This hash is used as a key to perform a cache lookup. The cache can be implemented on disk in a directory, where files are named with the key. On a cache hit, the backend loads the entry from the cache and returns it. On a cache miss the optimizations and code generation must proceed, and the resulting object file is committed to the cache before being returned to the linker.

This scheme is relatively coarse grain, since it operates at the module level even if a single function is modified. However it naturally fits into LLVM, and provides the guarantee that the final binary is the same whether an incremental build was involved or not. Note that the hash computed for each module's IR reflects any changes to command-line options or profile data affecting its generated code.

For a distributed build, the global analyses results relevant to each module will be serialized out (see Section 6). A backend job only needs to be scheduled if its IR file or analysis results change.

Section 8.3 shows how effective this scheme is in practice, allowing use of ThinLTO during incremental development.

4. Function Importing

The key transformation enabled by ThinLTO global analyses is function importing, in which only those external functions deemed likely to be inlined are imported into each module. This minimizes the memory overhead in each ThinLTO backend, while maximizing the most impactful CMO opportunity: inlining. The IPO transformations are therefore performed on each module extended with its imported functions. As introduced in Section 2.2, this optimization is split into two parts: an analysis that is performed purely over the summaries during the second phase (thin link), and a transformation during the third phase (ThinLTO backend) where the IR is actually modified to perform the import. This section details these two parts and their implementation.

In the second phase (thin link), the merged call graph is traversed to identify functions likely to benefit from importing into each module. These are typically small hot functions that are profitable to inline at their callsites. Each chain of calls is followed until no profitable imports are found. The profitability to import a function is controlled by a threshold (currently on the function size). The threshold decays as each call chain is traversed, as callees further away from the original importing module are less likely to actually be inlined into a function within the importing module.

Additionally, when making function importing decisions, symbols referenced by imported functions will be marked *exported*. This has implications for local symbols, as detailed in Section 4.1, and on global optimizations such as internalization, described later in Section 5.1.

During the third phase (ThinLTO backend), the module being compiled imports the identified functions from their defining module, using the paths in the combined index's module path symbol table. To reduce I/O, functions to be imported from each module are batched, and each source module is opened exactly once for each parallel backend importing from it. Other strategies to reduce I/O overhead are employed in our implementation, such as encoding in each IR file a table of offsets into that file from where to load the IR of each constituent function body.

The function importing transformation occurs very early in each backend, so that inlining and other IPO can take advantage of the extended module. The imported function symbols are marked so that their definitions can be dropped after IPO, preventing further compile time effects.

4.1 Promotion of Symbols with Local Linkage

When an imported function includes a reference to a local symbol in the original module, that symbol must be promoted to global scope so that it can be referenced from the importing module. Additionally, in order to disambiguate promoted locals from other promoted locals or a global with the same name, the promoted symbol must be renamed. However, the symbol will need to be renamed consistently in multiple independent ThinLTO backend processes: the original defining (exporting) module, and all modules importing its reference. Therefore, the renaming scheme should apply an identifier that is associated with the source module in the combined summary index. We simply append the SHA-1 hash of the source module's IR file, which is recorded in the combined index as will be described in Section 7.1.

5. ThinLTO Cross-Module Optimizations

Besides function importing/inlining, other global analyses and optimizations (including WPO) can be performed with ThinLTO. As described in Section 2.2, these optimizations are split into two parts. The first part is the index-based global analysis, performed during the thin link phase, for which the results are recorded in the index. The second part is the transformation, which is applied independently on the IR at the start of the ThinLTO backend, using the information recorded in the index. This section describes some of these global analyses and optimizations.

5.1 Internalization

In regular LTO, after all IR is merged into a single monolithic module, the compiler has visibility into all symbol definitions and uses within the IR. Any symbol that does not

need to be visible outside of the LTO merged module can be *internalized*, meaning it is transformed from a global symbol into a local one. In LLVM this is done by changing the linkage type.

There are several advantages to internalization. The first is that the symbol definition can be discarded by the compiler if there are no references, e.g. after inlining, resulting in a smaller resulting object file and binary. For example LLVM is very likely to inline a local function that has a single callsite, since the function can be discarded immediately afterward. Another advantage is that all uses of a symbol are known, which can result in more accurate analyses and more aggressive optimizations. Local functions are not bound by the ABI: calling conventions can be adjusted at will, unused function parameters can be removed. For local variables, the compiler may conclude that the address is not taken, enabling more accurate alias analyses.

In ThinLTO, we don't have a single monolithic module on which to perform internalization. Additionally, due to function importing, we will create new external references for each module, as described earlier in Section 4. However, during the second phase (thin link), using the reference graph and the result of the function importing analysis, we flag for internalization any global symbol that is only referenced from within its defining module.

The amount of internalization that can be applied with ThinLTO is inherently smaller than in regular LTO, since keeping the modules separated induces cross-module references. However, with linker dead stripping during the final native object link, the resulting ThinLTO text size is nearly the same as regular LTO, as will be shown in Section 8.5.

5.2 Weak Symbol Resolution

For symbols with weak linkage, the linker will keep one *prevailing* copy, discarding the remaining *preempted* copies. While it would be legal to emit all copies of weak symbols in the ThinLTO backends, and let the final native object link select one, we can reduce compile time through the backend by marking the preempted copies for deletion after inlining.

Additionally, in LLVM there are two classes of weak linkage [25]: *weak* linkage which means there may be a use outside of the module and the symbol definition is always emitted into the native object, and *linkonce* linkage which means that the symbol definition may be dropped during compilation if there is no reference within that module (e.g. after inlining). As a result, if there are any exported references of a linkonce symbol, the *prevailing* copy must be promoted to weak linkage to ensure that it is retained to satisfy any exported references at link time.

5.3 Indirect Call Promotion

As will be described in Section 7.2.2, profiled references from indirect callsites are recorded in the function summary. During the thin link, potental indirect call targets that may be promoted and further inlined are marked for importing into the caller's module. The actual indirect call promotion transformation happens in the backend compilation phase.

6. Distributed Build Implications

To support distributed builds, the results of the global analyses must be serialized out to disk. However, rather than transfer the entire combined index for the application to each remote build machine, for each module we can simply emit a subset of the combined index containing results for that module's backend compilation. This subset includes the summaries for the module's own defined symbols as well the functions it should import, along with a module path symbol table entry for each module being imported from. This communicates the importing decisions to the backend, along with the results of the global analyses recorded in the index as described in Section 5.

Additionally, for distributed build systems without a network file system, the intermediate object files containing the functions to import must be staged to the remote machine's local storage. To aid this process, the thin link phase is configured to emit plain text lists of intermediate object files that are additional inputs for each module's backend invocation.

7. Implementation

ThinLTO is currently implemented in the upstream clang and LLVM compiler [23]. To enable ThinLTO, simply add the -flto=thin option to compile and link. E.g.

```
% clang -flto=thin -O2 file1.c file2.c -c
% clang -flto=thin -O2 file1.o file2.o
```

Currently, the gold, ld64 and lld linkers support ThinLTO links of LLVM IR. By default the linker will launch the third phase (ThinLTO backend) in parallel threads, passing the resulting native object files back to the linker for the final native link. As such, the usage model is the same as non-LTO and no change to the existing build system is required. All LLVM utilities support ThinLTO intermediate object files.

This section describes LLVM-specific aspects of the implementation.

7.1 LLVM Bitcode Representation

When either regular LTO or ThinLTO is invoked for a clang/LLVM compile step, the resulting object file contains the LLVM IR encoded in the bitcode file format [24]. The bitcode format is essentially a binary encoding of structured data, organized as nested blocks containing data records.

7.1.1 Module Bitcode Files

For ThinLTO, the module bitcode files produced by the first phase (compile) require two additions. First, the module summary described in Section 2.1 is encoded in a new block that can be easily read without parsing the rest of the IR.

Second, to support incremental relinks of the program, a hash of the serialized bitcode is computed on the fly and the

resulting SHA1 identifier is recorded in the file. Section 3 detailed the incremental build support.

7.1.2 Combined Index

The combined index created in the second phase (thin link) can also be serialized out to a standalone bitcode file. While the in-memory index is typically used when the backend threads are launched from the linker, serializing it out is useful for debugging, and also for the distributed build scheme presented in Section 6.

The combined index described in Section 2.2 contains a summary block aggregated from the individual module summary blocks, and a table of module paths. It also contains a symbol table of global unique identifiers (GUID) for the values in the summary. The GUID is the MD5 hash of an identifier formed from the original symbol name, with the source file path appended for local values. This is not only more compact than including the full symbol names, but also aids integration with indirect call profiles, as will be described in Section 7.2.2.

7.2 Integration with Profile Guided Optimization (PGO)

While profile feedback is not required for ThinLTO, it is complementary. The index is designed to integrate with profile data when provided for PGO [4].

7.2.1 Direct Call Edge Profiles

As noted in Section 2.1, profile data is encoded in the summary: call edges can be marked *hot* or *cold*. This information is propagated along with the rest of the summary information into the combined summary index. It is used to help guide function importing decisions, as will be shown in Section 8.

7.2.2 Indirect Call Profiles

Profile-based indirect call promotion [1] is an effective technique not only for reducing the frequency of indirect branches, but more importantly for enabling inlining of indirect call targets that can't be statically resolved. Indirect calls are particularly prevalent in C++ due to virtual functions. The LLVM compiler supports value profiling of indirect call targets. On the subsequent PGO compile, hot indirect calls with up to two frequent targets are transformed into a series of direct calls to those targets, guarded by target address checks, and a fall-through indirect call.

In the LLVM implementation, the profile encodes the profiled targets using a GUID which is the MD5 hash of the original target callee name, with the source file appended for local functions. The indirect call value profile data is attached to indirect call instructions in the IR as metadata [25]. The indirect call promotion pass will use this metadata to create the guarded direct calls for hot indirect calls.

Because the possible targets are identified by a hash of their name, the transformation is only possible for target functions available in the module (a mapping from hash to function name is required). Also, most of the benefit from indirect call promotion comes from being able to inline the body of the new direct called functions. For monolithic LTO, all the functions are available. For ThinLTO, the target functions of indirect calls need to be made available through the function importing transformation presented in Section 4.

When the module summary is being built during the first phase (compile), indirect call value profile metadata attached to indirect calls in the IR are translated into additional direct call edges to each associated GUID. Note that this GUID is the same global identifier used for symbols in the combined index, as described in Section 7.1.2. The combined index will include GUID-indexed summaries for any profiled targets that are defined in the application's IR. These profile-based indirect call edges look exactly the same as direct call edges, and are therefore imported in the same way. The subsequent indirect call promotion pass in the third phase (ThinLTO backend) will be able to promote any profiled indirect call target definition that was imported to the module, which can then be inlined into the new direct call.

8. Performance Evaluation

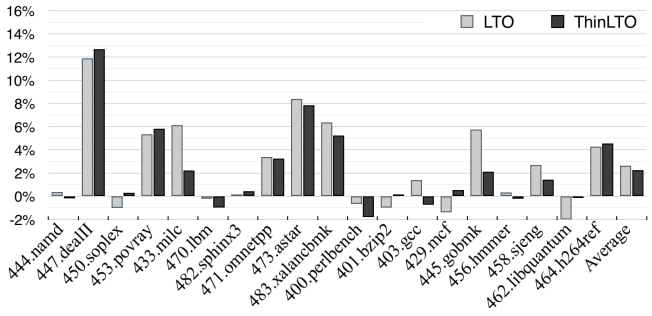
Using the clang/LLVM (v4.0), and GCC (v7.0) compilers built from recent (August 2016) upstream development sources, we collected a variety of runtime performance and build statistics for ThinLTO and regular LTO. We compare against GCC as it has a well-tuned and documented open-source LTO implementation.

Unless stated otherwise, ThinLTO is configured to only import called external functions that have less than 100 instructions recorded in the summary for that callee. Additionally, as described in Section 4, the instruction threshold decays as the call chain is traversed. The default decay factor is 0.7 (so the callees of the first level of imported functions have an instruction limit of 70, and so on). When PGO data is available, edges marked *hot* as described in Section 7.2.1 are given a higher limit of 300 instructions and a decay factor of 1.0.

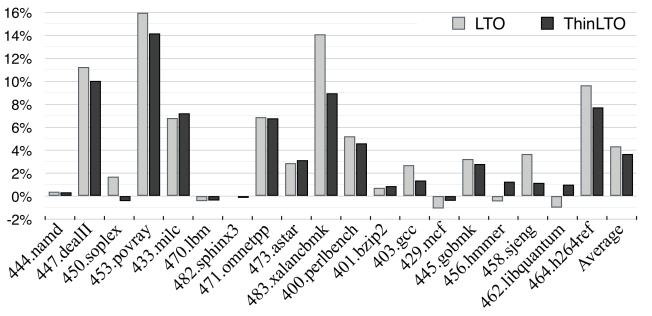
8.1 Runtime Performance

Performance results were collected for the C/C++ SPEC cpu2006 benchmarks on an 8-core 2.6GHz Intel Xeon E5-2689. The benchmarks were built with LLVM in three different configurations: plain -O2, LTO -O2, and ThinLTO -O2. Each benchmark was run in isolation five times, and results are shown for the average of three runs after discarding the highest and lowest result.

Figure 2a shows the results when compiling without PGO. While tuning is still in progress, ThinLTO already performs well compared to LTO, in many cases matching its performance. In a few cases ThinLTO even outperforms LTO, which is possible because the reduced scalability of LTO requires use of a slightly less aggressive optimization pipeline than either non-LTO or ThinLTO.



(a) Without Profile-Guided Optimization (PGO).



(b) With Profile-Guided Optimization (PGO).

Figure 2: SPEC cpu2006 Performance Improvement over -O2 (No LTO) for ThinLTO and LTO.

	Functions Imported Per Module			Modules Imported From Per Module		Total
Benchmark	Avg	Max	Inlined	Avg	Max	Modules
444.namd	1.27	8	42.86%	0.82	5	11
447.dealII	7.59	81	86.02%	5.52	25	108
450.soplex	6.29	32	47.12%	3.47	18	63
453.povray	19.38	192	46.64%	8.82	51	99
433.milc	4.15	30	35.00%	2.38	16	68
470.lbm	3.50	7	57.14%	0.50	1	2
482.sphinx3	10.08	33	63.09%	5.25	18	44
471.omnetpp	20.37	55	48.91%	9.10	17	84
473.astar	2.73	21	70.00%	1.27	8	11
483.xalancbmk	12.83	116	56.03%	6.49	30	695
400.perlbench	52.38	108	49.80%	12.94	18	50
401.bzip2	2.57	8	22.22%	1.14	2	9
403.gcc	50.51	329	47.62%	17.75	54	146
429.mcf	1.18	5	61.54%	0.91	4	11
445.gobmk	28.98	104	34.75%	11.68	25	67
456.hmmer	5.73	37	55.81%	4.02	13	57
458.sjeng	5.31	28	64.71%	2.31	10	19
462.libquantum	10.08	36	29.77%	4.62	14	16
464.h264ref	9.00	81	41.36%	4.75	27	42

Table 1: Importing statistics for cpu2006 (no PGO)

Figure 2b shows the results when compiling all configurations with PGO using the same profile data. As described in Section 7.2, ThinLTO uses the indirect call value profiles to enable cross-module indirect call promotion via importing. The cross-module indirect call promotion improves ThinLTO performance by close to 6% on 453.povray, and around 5% on 483.xalancbmk.

The few gaps between ThinLTO and LTO performance will be reduced by enhancements to summary-based optimization, such as identifying constant values to enable interprocedural constant propagation.

Table 1 shows some statistics on how many functions are imported into each module, and how many unique modules those imported functions are sourced from, as well as what percentage of imported functions are inlined into the importing module. While on average most modules import functions from less than 10 other modules, there are cases where a module may import from around 50 other modules. This illustrates the importance of being able to import just those functions likely to benefit from inlining, in order to reduce the memory overhead of each backend. The results for the percentage of imported functions inlined into the importing module show that there is room for further tuning of importing heuristics, to avoid unnecessary overhead from unprofitable imports.

We have also collected performance data using both smaller and larger importing thresholds. The full results are not shown for brevity, but it appears that the default thresholds are enabling most of the available benefit. Doubling the threshold from 100 to 200 results in similar performance for most benchmarks, with speedups of around 2% in 447.dealII and 483.xalancbmk. Reducing the threshold in half to 50, however, results in significant degradations in a number of benchmarks, including 5% in 453.povray and 6.7% in 433.milc. While there are benefits to tightening the threshold for importing, which should reduce the build overhead and also enable more internalization, clearly it must not be done indiscriminately.

To determine how much of the ThinLTO improvement comes from importing, we also collected performance data with importing disabled, both with and without PGO. Most of the performance gain of ThinLTO over non-LTO is lost, with the notable exception of 473.astar, which retains about 90% of the gain. This is due to internalization performed during the thin link, as described in Section 5.1, which discovers that several functions are never accessed outside of their defining module. The internalization enables more aggressive inlining.

Additionally, we evaluated the runtime performance of the clang binary itself. We have measured 5-8% faster performance when clang is built with ThinLTO, compared to a non-LTO build, very similar to the performance from clang built with regular LTO.

8.2 Build Performance

Critically, due to the scalable design of ThinLTO, this performance is achieved with a build time that stays within a non-LTO build scale.

This section analyzes build performance, comparing both the build time and the memory consumption of ThinLTO to LLVM LTO and GCC LTO. While LLVM LTO does not have separate WPA and CMO optimization phases, GCC

Application	Files (#k)	Size (MB) -g0	Size (MB) -g	Nodes (#k) F.	Nodes (#k) GV.	Nodes (#k) GA.	Nodes (#k) **Tot.**	Edges (#k) Calls	Edges (#k) Refs	Edges (#k) **Tot.**
Clang	1.9	217	3554	76	91	1.6	**169**	216	198	**414**
Chromium	17.8	706	7544	559	295	3.7	**858**	1328	1075	**2403**
Ad Delivery	13.8	1073	7469	1109	760	84	**1953**	2251	2605	**4856**

Table 2: Number and sizes of IR files for large real-world applications used in build measurements, and statistics for thin link graph: number of nodes (Functions, Global Variables, Global Aliases) and edges (Calls and References).

splits these into separate processes. Therefore we can compare the ThinLTO thin link phase to the GCC WPA phase: these are the serial WPA phases of each solution. Later we compare the ThinLTO parallel backend phase to GCC LTRANS: these are the CMO and codegen phases of each solution.

Three large real-world applications of increasing size have been used to perform the comparison:

1. The Clang compiler [22]: This is an open-source code base including multiple millions of lines of modern C++.
2. Chromium [21]: This is an open-source Web Browser.
3. Ad Delivery: This is an internal Google application running in the datacenter that serves targeted ads.

Table 2 indicates the size of these applications. In addition to the number of IR files input to the link, the total size of the IR input files for both a -g0 (no debug info) build and a -g (debug info) build are shown for LLVM ThinLTO. The increase in total IR size from the ThinLTO summaries is small, for example 0.8% for clang with -g0 (even less for -g2). We also report the size of the static graph from the thin link phase in terms of number of nodes and number of edges. The applications range from 169k nodes and 414k edges to 1.9M nodes and 4.9M edges, which are the metrics that impact the serial phase processing (thin link and WPA).

Build times and memory overhead were collected on a 16 core (32 logical) 2.9GHz Intel Xeon CPU ES-2690 with 64GB memory, running Linux and using the gold linker.

To evaluate the build memory overhead, we measured the peak difference between the *resident set size* and the *shared memory* as reported by the Linux Kernel for the process. The shared memory accounts for read only mmapped files that the kernel greedily keeps in RAM (even on unmmap) as long as there is available memory. This memory is included in the resident set size even though it does not incur any memory pressure on the system.

Figure 3 shows the time and memory comparisons together for the link of Chromium using 32 threads. Note that LLVM LTO optimizes in a single thread, and only the codegen is parallel. This graph illustrates how much smaller the ThinLTO memory footprint is, and how much faster it completes the link compared to either LLVM LTO or GCC LTO.

Figure 4a shows the peak memory required for the serial steps in ThinLTO and GCC LTO. As expected, ThinLTO's serial step uses very small amounts of memory, scales well with the size of the program, and isn't affected by debug information. The results show that GCC LTO is not usable

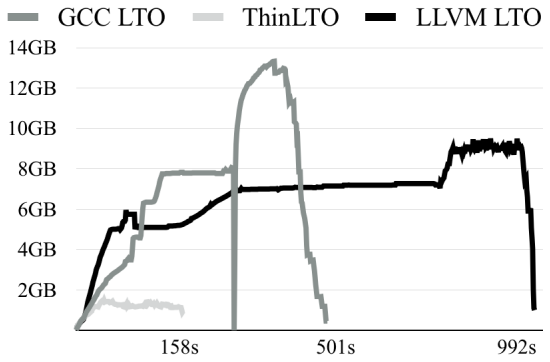
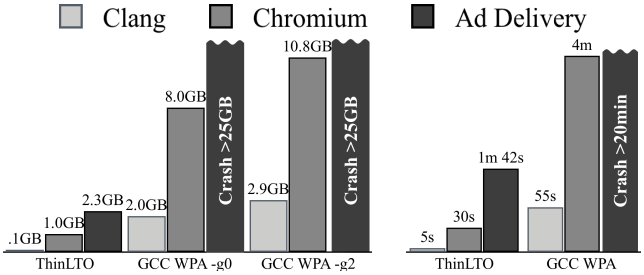


Figure 3: Time versus Memory for the Chromium link. All configurations utilize 32 threads (only affects code generation for LLVM LTO). The spike towards zero on the GCC curve corresponds to the transition from WPA to LTRANS.



(a) Peak Memory (ThinLTO thin link is not affected by debug information).

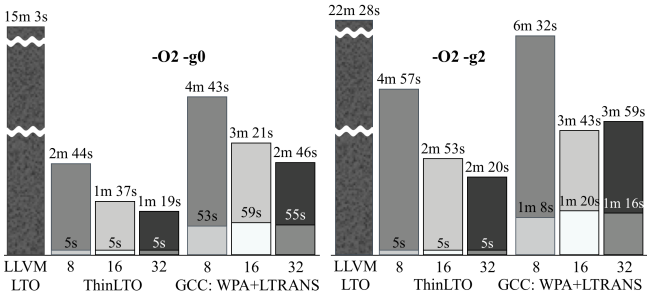
(b) Time (-g0).

Figure 4: Serial phase measurements for ThinLTO (thin link) and GCC (WPA configured with 32 partitions), for the three programs described in Table 2.

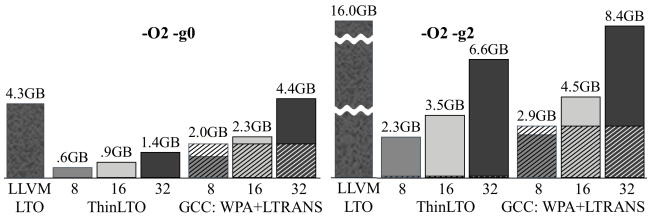
for large applications like Ad Delivery, as it cannot complete the link. Figure 4b shows the corresponding timings which illustrate how ThinLTO can handle large applications.

Figures 5a and 6a show the time required to link the IR files of Clang and Chromium, respectively. This includes both the serial LTO step and the backends for ThinLTO, LLVM LTO and GCC LTO, using varying amounts of parallelism (except for LLVM LTO). Because the ThinLTO serial step (the second phase thin link) is so fast, and the actual optimization work is done in parallel backends, ThinLTO is much faster than LLVM LTO in particular. GCC LTO is faster than LLVM LTO, but slower than ThinLTO and doesn't scale as well due to the longer serial step. Ad Delivery is not shown as LLVM LTO did not complete after 2h, and as shown in Figure 4 GCC LTO does not complete the serial WPA step.

Figures 5b and 6b compare the peak memory consumption during the link. The solid bars are the peak memory for just the backend phases, whereas the overlaid hashed bars show the peak for the serial phases of ThinLTO and GCC. Note that the overall peak memory for GCC is from the serial WPA phase under lower parallelism. LLVM LTO cannot successfully link Chromium with -g because of the size

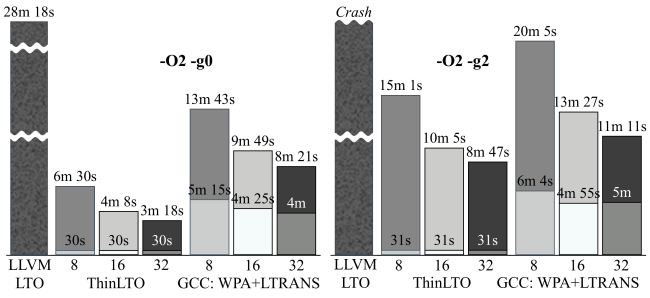


(a) Time for the link. The inner bar represents the part of the serial phase (thin link or GCC WPA) in the total.

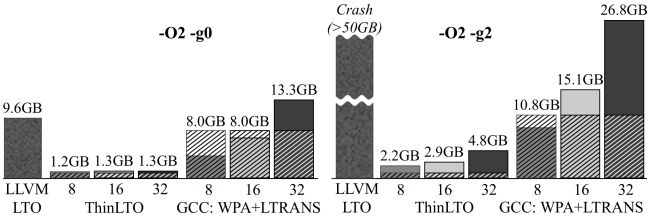


(b) Peak Memory for the link. The hashed bars represent the serial phase memory from Figure 4a (too small to see in ThinLTO), and the solid bars are the backend phase.

Figure 5: Clang link statistics (WPA + Backends) for LLVM LTO, ThinLTO and GCC for 8, 16, and 32 threads, without and with debug information.



(a) Time for the link. The inner bar represents the part of the serial phase (thin link or GCC WPA) in the total.



(b) Peak Memory for the link. The hashed bars represent the serial phase memory from Figure 4a (too small to see in ThinLTO), and the solid bars are the backend phase.

Figure 6: Chromium link statistics (WPA + Backends) for LLVM LTO, ThinLTO and GCC for 8, 16, and 32 threads, without and with debug information.

induced by the debug information. ThinLTO exhibits good memory scaling. Only ThinLTO can complete Ad Delivery (1m42s thin link and 5m43s backends), peaking at 3.6GB.

8.3 Incremental Build Performance

One of the major features of ThinLTO is the ability to perform incremental builds, as presented in Section 3. Figure 7 illustrates the effectiveness of incremental builds by comparing LTO, ThinLTO, and No-LTO (-O2) when building Clang in three different situations:

1. The full time for building from a clean build directory.
2. The developer changes the implementation of a frequently referenced function in a widely used header, which forces the rebuild of a large number of files. This is the time to recompile these source files (first phase) and perform the link (second and third phases).
3. The developer changes the implementation of an infrequently called function in an implementation file, so the incremental build time should be fast.

The clean build shows that ThinLTO build time is very comparable to a non-LTO build, while regular LTO is significantly slower. With a warm build cache, even with various types of modifications, ThinLTO provides an incremental link-time very close to a non-LTO build. This shows how ThinLTO can replace the non-LTO build in day-to-day development. It can even be faster than a non-LTO build: e.g. a header file modification will not affect the IR of all including modules, and in those cases ThinLTO will use the cached

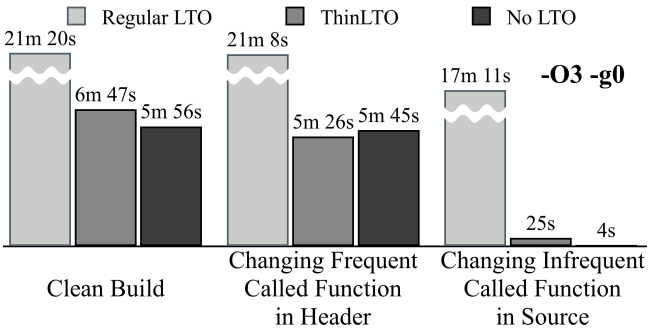


Figure 7: End-to-end incremental build times for Clang LLVM LTO, ThinLTO and No-LTO (-O3) with -j32.

object, skipping many optimizations and codegen. Regular LTO is not friendly with incremental builds.

At the same time, the developer gets a significant performance improvement on the resulting clang binary, as shown at the end of Section 8.1.

8.4 Distributed Build Performance

Using experimental support for ThinLTO distributed builds added to the Bazel build system [10], we compared performance of ThinLTO and regular non-LTO builds of Ad Delivery on a cluster of remote machines. This application has a complicated build flow, with intermediate tools that are built and generate inputs for subsequent build actions.

Figure 8 shows the results of the end-to-end builds without caching, for various values of -j*N*. Note that *N* specifies

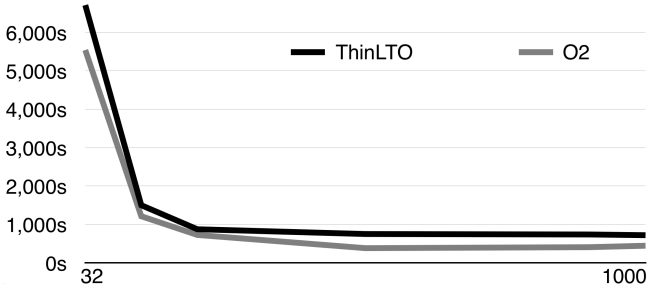


Figure 8: Distributed end-to-end build times for Ad Delivery -O2 and ThinLTO with various -j*N* allowed parallelism values (best of 3 runs).

a limit on the amount of parallelism, which is a hint to the build system. The actual parallelism may vary depending on the load of the shared cluster. The graph illustrates Amdahl's law in effect between 200 and 500 nodes, where the critical path for this build is reached.

The results show that ThinLTO benefits from distribution as much as a regular non-LTO build, despite the intermediate thin link serial step and the overhead of distributing the additional files. As noted earlier, neither LLVM nor GCC LTO can successfully build Ad Delivery in a reasonable amount of time or memory, much less benefit from distribution.

8.5 Binary Size

There are several factors affecting ThinLTO binary sizes. While ThinLTO importing does temporarily create duplicate copies of functions, those that are not inlined are subsequently deleted. However, ThinLTO must promote to global scope any local symbols that are exported by an imported function. Linker dead stripping compensates by dropping any exported references that are not kept via inlining. Also, as described in Section 5.1, the amount of internalization is inherently smaller in ThinLTO than in LTO, as there are remaining cross-module references.

Despite these factors, the sizes of the binary and text for a non-debug compile of Clang, as well as the text size of a debug build, are nearly identical to that of a full LTO build. The debug binary size is much larger (622MB LTO vs 2429MB ThinLTO) due to known weaknesses in handling of imported LLVM debug metadata types. Prototype improvements have already demonstrated a significant reduction to 1335MB, which is very close to the non-LTO debug binary size of 1203MB.

9. Future Work

The ThinLTO framework provides opportunities for many additional types of CMO and WPA. For example, we are currently investigating global dead symbol stripping based on the existing summary reference graph. This optimization improves the build time as it prunes the size of the graph, but can also improve the accuracy of other analyses. Another possible optimization is *moving* functions instead of importing them: for instance if a function is only called from a single module, moving it enables internalization in this module. A related idea is a hybrid model, which would sometimes merge small tightly coupled modules to enable additional internalization.

Some new optimizations will require additional types of summary data. For example, adding information in the summaries about the possible values or ranges of values that a global variable can take could be used to perform interprocedural constant propagation. The key to many future WPA based optimizations will involve making additional LLVM analyses and tranformations summary-aware.

We are currently extending the summaries with enough information to rebuild the full class hierarchy at link-time as well as the possible overloads for each virtual call. This would enable C++ global devirtualization as well as Control Flow Integrity [26].

The dependency tracking using a hash at the module level is quite coarse-grain. We identified the possibility to track dependencies with a finer grain on the callgraph, as described in [5]. However we anticipate that the overhead may lead to some interesting tradeoffs in this area.

Improvements to the representation and importing of LLVM debug metadata will reduce *-g* memory and binary overheads from ThinLTO. Additional memory overhead reductions will come from fine-tuning the importing algorithm, particularly with profile data.

10. Conclusion

ThinLTO is a new framework for CMO, that was designed to be both scalable and allow for incremental compiles. It achieves this by including summary data in the intermediate object files, and only performing fast summary based WPA in the serial step, without reading or writing IR. All CMO is applied in fully parallel backends, on modules extended using function importing. This decoupling of the WPA from the CMO is not only efficient, but also enables stable incremental compilation which is not achievable with traditional LTO, and integrates well with distributed build systems.

Performance results for SPEC cpu2006 benchmarks show that ThinLTO can achieve most of the performance of regular LTO. At the same time, build time and memory data for three large real-world applications show that ThinLTO is smaller, faster, and more scalable than both LLVM LTO and GCC LTO.

All this makes ThinLTO a good candidate to be enabled by default and replace the standard non-LTO flow.

Acknowledgments

The authors thank Duncan P. N. Exon Smith for sharing pearls of wisdom with us during the course of this project, as well as Béatrice Creusillet and Sanjay Ghemawat, for their meaningful comments on this paper.

References

[1] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *Proceedings of the 10th European Conference on Object-Oriented Programming*, ECCOP '96, 1996.

[2] AMD. Using the x86 open64 compiler suite. *Advanced Micro Devices*, 2011. URL http://developer.amd.com/assets/x86_open64_user_guide.pdf.

[3] A. Ayers, S. de Jong, J. Peyton, and R. Schooler. Scalable cross-module optimization. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 301–312, New York, NY, USA, 1998. ACM. URL http://doi.acm.org/10.1145/277650.277745.

[4] R. Cohn and P. G. Lowney. Feedback directed optimization in Compaq's compilation tools for Alpha. In *Proceedings of the Second Workshop on Feedback-Directed Optimization, held in conjunction with MICRO-33*, pages 3–12, Nov. 1999.

[5] K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 58–67, New York, NY, USA, 1986. ACM. URL http://doi.acm.org/10.1145/12276.13317.

[6] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Softw. Pract. Exper.*, May 1991.

[7] J. W. Davidson and A. M. Holler. A study of a c function inliner. *Softw. Pract. Exper.*, Aug. 1988.

[8] Free Software Foundation, Inc. GCC, the GNU compiler collection, 2016 (accessed 29-November-2016). URL https://gcc.gnu.org.

[9] T. Glek and J. Hubicka. Optimizing real world applications with GCC link time optimization. *CoRR*, 2010. URL http://dblp.uni-trier.de/rec/bib/journals/corr/abs-1010-2196.

[10] Google. Bazel, 2015 (accessed 29-November-2016). URL https://bazel.build.

[11] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Houston, TX, USA, 1991. UMI Order No. GAX91-36029.

[12] Itanium C++ ABI. Vague linkage, 2001 (accessed 29-November-2016). URL http://mentorembedded.github.io/cxx-abi/abi.html#vague.

[13] R. Ju, S. Chan, F. Chow, X. Feng, and W. Chen. Open research compiler (ORC): Beyond version 1.0. In *Tutorial presented at the 11th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, Mar 2004.

[15] X. D. Li, R. Ashok, and R. Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2010.

[16] X. D. Li, R. Ashok, and R. Hundt. LIPO - profile feedback based lightweight IPO, 2013 (accessed 29-November-2016). URL https://gcc.gnu.org/wiki/LightweightIpo.

[17] Microsoft Corporation. Speeding up the incremental developer build scenario, 2014 (accessed 29-November-2016). URL https://blogs.msdn.microsoft.com/vcblog/2014/11/12/speeding-up-the-incremental-developer-build-scenario.

[18] S. Moon, X. D. Li, R. Hundt, D. R. Chakrabarti, L. A. Lozano, U. Srinivasan, and S.-M. Liu. Syzygy - a framework for scalable cross-module ipo. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, 2004.

[19] R. Muth, S. Debray, S. Watterson, K. D. Bosschere, and V. E. E. Informatiesystemen. Alto: A link-time optimizer for the Compaq Alpha. *Software - Practice and Experience*, 31: 67–101, 1999.

[20] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. Research Report, 1992.

[21] The Chromium Projects. Chromium, 2016 (accessed 29-November-2016). URL https://www.chromium.org/Home.

[22] The LLVM Foundation. clang: a C language family frontend for LLVM, 2016 (accessed 29-November-2016). URL http://clang.llvm.org.

[23] The LLVM Foundation. The LLVM compiler infrastructure, 2016 (accessed 29-November-2016). URL http://llvm.org.

[24] The LLVM Foundation. LLVM bitcode file format, 2016 (accessed 29-November-2016). URL http://llvm.org/docs/BitCodeFormat.html.

[25] The LLVM Foundation. LLVM language reference manual, 2016 (accessed 29-November-2016). URL http://llvm.org/docs/LangRef.html.

[26] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 941–955, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL http://dl.acm.org/citation.cfm?id=2671225.2671285.

[27] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *International Symposium on Signal Processing and Information Technology*, pages 7–12, 2005.