Lightweight Feedback-Directed Cross-Module Optimization

Xinliang David Li, Raksit Ashok, Robert Hundt

Google
1600 Amphitheatre Parkway
Mountain View, CA, 94043
{davidxl, raksit, rhundt}@google.com

Abstract

Cross-module inter-procedural compiler optimization (IPO) and Feedback-Directed Optimization (FDO) are two important compiler techniques delivering solid performance gains. The combination of IPO and FDO delivers peak performance, but also multiplies both techniques' usability problems. In this paper, we present LIPO, a novel static IPO framework, which integrates IPO and FDO. Compared to existing approaches, LIPO no longer requires writing of the compiler's intermediate representation, eliminates the link-time inter-procedural optimization phase entirely, and minimizes code re-generation overhead, thus improving scalability by an order of magnitude. Compared to an FDO baseline, and without further specific tuning, LIPO improves performance of SPEC2006 INT by 2.5%, and of SPEC2000 INT by 4.4%, with up to 23% for one benchmarks. We confirm our scalability results on a set of large industrial applications, demonstrating 2.9% performance improvements on average. Compile time overhead for full builds is less than 30%, incremental builds take a few seconds on average, and storage requirements increase by only 24%, all compared to the FDO baseline.

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

General Terms Performance

Keywords Inter-procedural, Feedback-directed, Cross-module, Optimization

1. Introduction

A static compiler's ability to optimize code is limited by the scope of code it can see. Typically, compilers translate one source file at a time, operating at function granularity. Most compilers, often at higher optimization levels, start to gradually enable intra-module inter-procedural optimizations. For languages supporting independent compilation of separate source files, such as C, C++, or Fortran, source boundaries become an optimization blocker.

For example, consider the artificial code in Figure 1 with two source files `a.c` and `b.c`. While compiling `a.c`, the compiler has no knowledge of the body of function `bar()` and will have to emit two standard calling sequences to `bar()`, passing two parameters to each. If `bar()` had been inlined into `foo()`, across the module boundary, the body of `foo()` could have been reduced to a simple `return 0` statement.

```
a.c:
  int foo(int i, int j) {
     return bar(i,j) + bar(j,i);
  }

b.c:
  int bar(int i, int j) {
     return i-j;
  }
```

Figure 1. Simple example to illustrate benefits of IPO

Over the years, a large array of inter-procedural (IP) optimizations has been developed. The typical techniques are inlining and cloning, indirect function call promotion, constant propagation, alias, mod/ref, and points-to analysis, register allocation, register pressure estimation, global variable optimizations, code and data layout techniques, profile propagation techniques, C++ class hierarchy analysis and de-virtualization, dead variable and function elimination, and many more.

To study the effectiveness of some of these transformations, we used the open64 [19] compiler, release 4.1 (SVN revision r1442), and compiled SPEC2000 INT in a set of experiments. All experiments were run on an Intel Pentium 4, 3.4 GHz, with 4GB of memory. In each experiment, we disabled one inter-procedural optimization pass and measured the expected performance degradations. Since there are interactions between the various passes' performance contributions, this study can only give an approximation.

Our baseline is compiled at -O2 with IPO and feedback directed optimization (FDO). We evaluated the eight inter-procedural passes inlining, indirect call promotion, alias analysis, copy propagation, as well as function reordering, class hierarchy analysis, dead code analysis, and dead function elimination. Early inlining (at the single module level) remained enabled for all experiments. Only the first four inter-procedural optimizations showed measurable performance impact on these benchmarks, summarized in Figure 2. Correspondingly, we omit the results for the other four optimizations.

We find that two of the more important IPO passes are inlining, as turning it off results in a 17% performance degradation overall, with 252.eon degrading by 70%, as well as indirect function call promotion (3% degradation overall when turned off). Correspondingly, we focused on these two in our initial implementation of LIPO. All performance numbers presented later result from improved performance enabled by these two passes.

Inlining improves performance by eliminating procedure call overhead, adding context sensitivity, and creating larger optimiza-

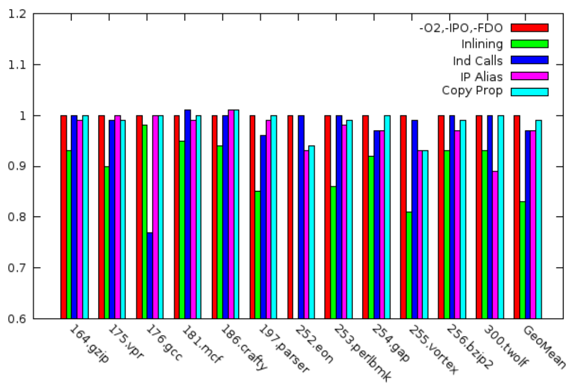

Figure 2. Effects of turning off individual IPO optimizations and analysis passes. Shown are relative regressions against normalized performance, 1.0 means no loss.

tion and scheduling regions. Added context sensitivity can improve alias and points-to information and enables more constant propagation, redundancy elimination, dead code and unreachable code elimination. It can also serve as an enabler for other major phases, such as the loop optimizer.

Indirect call promotion is an enabler for inlining. It replaces an indirect, hot call having a limited set of target addresses, with a cascade of tests guarding direct calls to those targets, plus a fall through branch containing the original indirect branch. The introduction of these direct calls enables further inlining (see also [1]).

1.1 Feedback Directed Optimization (FDO)

In this section we briefly describe feedback directed optimization (FDO) and study its impact on IPO. FDO imposes a dual build model. In a first *instrumentation build* the compiler inserts code into the binary, typically to count edges or to collect value profiles. The instrumented binary is run on a representative set of training input in a *training phase*. At the end of this execution, all collected edge counts and value information are written and aggregated in a profile database. In a second *optimization build*, the compiler uses the generated profile to make better optimization decisions. Many compiler phases can benefit from more exact information then estimated heuristics, e.g., the inliner and indirect call promotion, where knowing the exact distribution of targets is essential, the loop optimizer, where distinguishing loops with low or high trip count can be beneficial, and many other optimizations, at both high and low level.

IP optimizations strongly benefit from FDO. To illustrate the effects, we used the same open64 compiler to benchmark SPEC2000 INT and compared the baseline to builds using IPO and FDO turned on, individually and combined. In these experiments, plain FDO actually decreased performance slightly, indicating that our open64 version was not tuned well towards this set of benchmarks. We believe this only strengthens our argument, as less of the typical SPEC specific optimizations were perturbing the results. Our claim that IPO needs FDO is supported by the winning 12% performance (p) increase in Table 1, which almost doubles the effects of plain IPO alone.

1.2 Existing IPO Frameworks

In Section 2 we discuss some of the existing inter-procedural compilation frameworks in more detail, in particular HLO [2, 3], the old high-level optimizer from HP, its successor SYZYGY [16],

Experiment	SPEC score	Improvement
-O2	12.69	
-O2 -FDO	12.60	-1%
-O2 -IPO	13.46	6%
-O2 -FDO -IPO	14.27	12%

Table 1. $p(FDO + IPO) > p(FDO) + p(IPO)$

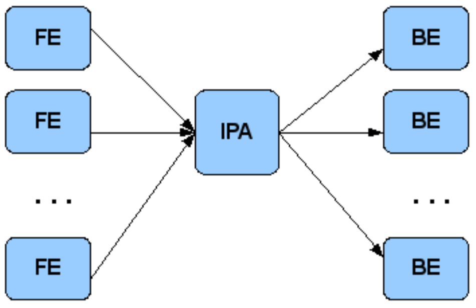

Figure 3. Traditional IPO model

the open source frameworks open64, gcc's *-combine* feature, gcc's LTO framework, and LLVM. We find that all existing infrastructures represent a variation of a standard IPO model, as presented in [16]. This model distinguishes three major phases, a front-end phase, an IPO phase, and a back-end phase.

In the parallelizable front-end phase, the compiler performs a reasonable amount of optimizations for code cleanup and canonicalization. It may also compute summary information for consumption by IPO. This phase writes the compiler intermediate representation (IR) to disk as fake ELF object files, which allows seamless integration into existing build systems.

The IPO phase is typically executed at link time. IPO reads in the fake object files, or parts of them, e.g., sections containing summary data. It may perform type unification and build an interprocedural symbol table, perform analysis, either on summaries, IR, or combined IR, make transformation decisions or perform actual transformations, and readies output for consumption by the back-end phase.

The parallelizable back-end phase accepts the compiler IR from IPO, either directly out of memory or via intermediate fake object files, and performs a stronger set of scalar, loop, and other optimizations, as well as code generation and object file production. These final object files are then fed back to the linker to produce the final binary.

All these tasks are points of distinction between the various frameworks. What all these frameworks have in common is that they write the compiler IR to disk, putting pressure on disk and network bandwidth. Because the compiler IR usually contains more information than a simple object file, e.g., full type information, IR files are typically larger than regular object files by factors ranging on average from 4x to 10x, with potential for pathological cases.

At IPO start, the IR files have to be read/mapped in, which can consume a significant amount of time, e.g., in the range of minutes, and even hours for large applications. While the front-end and back-end phase can be parallelized, IPO typically cannot, or only to a limited extent, representing a bottleneck with runtimes ranging again from minutes to hours.

Since the effects and dependencies of source changes aren't modeled, even insignificant code changes lead to full invocations of IPO and the complete back-end phase, which can also take very long for large applications. Overall, this design allows effective cross-module inter-procedural optimizations at the cost of very long overall compile/link cycles. We provide detailed examples for compile and link times for existing infrastructures in Section 2

Debugging of the IPA infrastructure is cumbersome, because of the many intermediate steps and the many files involved, as well as the high runtime of IPO itself. Maintaining debug information can be complicated for some compilers, depending on how they maintain or generate debug information (e.g., via callbacks into the compiler front-end). Combining FDO and IPO multiplies the usability problems, as now two IPO builds have to be performed, at least for compilers that expect identical control flow graphs during the FDO instrumentation and profile annotation phase.

1.3 Contribution

In this paper, we make the following contributions:

- We present our novel IPO framework, which seamlessly integrates IPO with FDO.
- We evaluate the infrastructures properties of our approach using the SPEC benchmark suite. We show that our approach performs an order of magnitude better than existing approaches in terms of compile time and storage requirements. Furthermore, our approach is amenable to distributed build systems.
- We demonstrate the immediate performance benefits from improved inlining and indirect call promotion, the only two optimizations we focused on in this paper.
- We confirm our results on a set of very large industrial applications, and provide further analysis results.

The rest of this paper is organized as follows. We first ask the reader for patience as we detail key design decisions of several existing IPO frameworks. These descriptions allow drawing a sharp contrast to our work, which we describe in Section 3. Readers familiar with existing frameworks can safely skip Section 2 and proceed directly to Section 3. We provide a thorough experimental evaluation in Section 4, before we conclude.

In this paper we always include cross-module optimizations when referring to IPO, unless noted otherwise. We use the term inter-procedural analysis (IPA) in cases where we refer to inter-procedural analysis only. While we used open64 for illustration above, all subsequent results reference our implementation based on gcc 4.4. All performance effects come from improved inlining and indirect call promotion only.

2. Related Work

Early work in inter-procedural optimization was done by Hall [13]. This work focused mainly on core IPO algorithms, such as call graph construction and inlining. Early studies of inlining have been done in [6, 9, 10] , and more recently in [5]. It became clear early on that even minor code changes would make a full IPO and back-end phase necessary. [8] tried to solve this problem by maintaining proper or approximated dependencies. While this certainly is a path to reducing the compile time of IPO, to our knowledge no commercially successful system has been deployed using such techniques. There have also been approaches to do IPO at link time on regular object files or fully built executables. A good example of such an approach is [18]. To a certain extent, dynamic optimizers could be considered inter-procedural optimizers, as they see the whole program at execution time. However, this field is far removed from the topic of this paper and we won't discuss it further here.

The earliest reference to feedback directed optimization was made by no other than Knuth [14], Ball and Larus had a seminal paper on optimal profile code insertion [4]. FDO is available in most modern compilers, with the notable exception of LLVM. A detailed description and evaluation of techniques to eliminate C++ virtual calls can be found in [1], class hierarchy analysis has been studied in [11].

We believe the most relevant and directly comparable pieces of work are the fully developed and deployed commercial and open source frameworks. In the next few sections, we detail key design choices made by HLO, an older inter-procedural optimizer from HP, its successor SYZYGY, the open source frameworks open64, gcc's *-combine* feature, gcc's LTO framework, and LLVM. We won't discuss other commercial compilers, e.g., from Intel, or Microsoft, but are confident that many of them implement a variation of the described general IPO model.

The HP High-Level Optimizer (HLO) [2, 3] maps in all input files at link time, and offers a compiler controlled swapping mechanism in order to scale to large applications, hoping that domain knowledge would beat the standard virtual memory management system. Code generation is an integral part of HLO, not parallelized, and a bottleneck in terms of compile time. On average, to compile SPEC2000 INT, HLO imposed a 2.5x overhead in compile time on full rebuilds, with up to 6x for larger benchmarks. For incremental builds, the compile time overhead factor is orders of magnitudes higher, as full IPO and backend phase have to be executed. HLO did not scale to the larger SPEC2006 INT benchmarks.

SYZYGY [16] is the successor of HLO and significantly improves compile time and scalability. It has a two-part IPO model. The first half operates on summaries only and makes most optimization decisions. During this time, no more than two IR files are opened at any given time. The second half of IPO consists of the inliner, which operates on summaries first to compute inlining decisions, before using actual compiler IR to make the transformations. In order to scale to very large applications, even with a constricted memory space, it maintains a pool of no more than a few dozen open IR files and algorithms were developed to minimize the file open and closing times [5]. At the end of IPO, final transformation decisions are written back to temporary IR files, and the back-end phase is parallelized over these files.

On average, to compile SPEC2000 INT, SYZYGY imposes a 2.3x overhead for full rebuilds at (backend) parallelism level 1, but only a 1.2x overhead at parallelism level 4. Again, for incremental builds, the compile time overhead factor is orders of magnitudes higher. The IR file to object file ratio was about 5x. Compiling a large shared library of a commercial database application, consisting of several thousand C input files, took around 4 hours for full builds, and 2 hours for incremental builds. The file overhead led to exceeding of the file size limit on the system, and build system changes were necessary to break this large library apart.

Open64's IPO maps in all IR files at link time. Its IPA phase runs comparatively fast, seeking to operate on summary data only. It writes back intermediate object files, and the back-end phase is parallelized over these files, similar to SYZYGY. One interesting design decision has been made for the inter-procedural symbol and type tables. Open64 produces a single intermediate object file containing all statically promoted and global variables, as well as the IP symbol and type table. In the parallelized backend phase, this file is compiled first, before all other intermediate files are compiled in parallel. All of these compilations pass a temporary IR file plus the symbol table file to the compiler. For large applications, the symbol table file can become many hundred MB in size, and as a result this design can significantly slow down compile time.

We used open64 (SVN revision 1442) to compile the C++ "Search" application presented in table 6 on a 4-core AMD Opteron machine, running at 2.2GHz, with 32GB of memory. Reading in and mapping of all input files required 6.5GB and took 59 minutes. Building other inter-procedural data structures required another 6.4GB of memory. The symbol table intermediate file ended up at 384 MB and took 53 minutes to compile. The resulting assembly file had 23 mio lines. The rest of the backend and code generation took many hours to complete, compiling an average of 5 files per minute per core (running 5 processes in parallel).

The recently started gcc LTO effort writes *fat* object files to disk in the front-end phase, containing both compiler IR and regular object file sections. This allows LTO to use the same object files for regular builds or IPO builds. LTO has a very narrow IPO phase, which works on summaries only and makes mostly grouping decisions, before parallelizing the back-end over these groups. The back-end is an extended gcc compiler, which accepts multiple IR files as a combined input and performs existing inter-procedural transformations. This is somewhat similar to the *-combine* support described below. At time of this writing, no scalability results were available.

LLVM [15] also has a traditional link time optimizer. It keeps the full compiler IR in core, alongside analysis data structures, such as use-def chains. Because of this design, the LLVM IR has been specifically optimized for memory footprint. In practice, LLVM is capable of optimizing mid-size programs (500K - 1M LOC) on desktop workstations. For example, compiling 176.gcc requires roughly 40MB of memory to hold in core. LLVM also performs code generation sequentially in core and produces one large assembly file. While efforts are underway to parallelize LLVM, in particular code generation, its current architecture is not capable of distributing to more than one machine [7].

The gcc C front-end supports the `-combine` option, allowing to combine sources into one compilation process in an all-in-memory model. This model differs from the general IPA approach, as users must manually partition the source files into sets, a labor intensive process which is unrobust against program evolution and obtrusive to the build system. The implementation is unloved by the gcc community because of its lack of robustness, falsely reported errors, and restriction to C.

3. Lightweight IPO

In this section we detail the design of our novel lightweight inter-procedural optimizer LIPO. The key design decisions can be summarized the following way:

- We seamlessly integrate IPO and FDO.
- We move the IPA analysis phase into the binary and execute it at the end of the FDO training run.
- We add aggregated IPA analysis results to the FDO profiles.
- During the FDO optimization build, we use these results to read in additional source modules and form larger pseudo modules to extend scope and to enable more intra-module inter-procedural optimizations.

We now discuss this design in detail with focus on the two key IP optimizations inlining and indirect call promotion.

Since IPO needs FDO to maximize its performance potential, integrating these two techniques becomes a logical design choice. Existing FDO users can get IPO almost for free by adding an option.

LIPO no longer needs an explicit inter-procedural optimizer to be executed at link time. Instead, the IPA analysis phase now runs at the end of the FDO training run, with negligible performance overhead. At this point, the analysis phase can see the complete results of the binaries' execution, in particular, all profile counters, debug and source information, as well as summary information, which may have been stored in the binary.

From this information, LIPO constructs a full dynamic call graph and performs a greedy clustering algorithm to determine beneficial groupings for inlining and indirect call promotion. The clustering information, and further analysis results, are stored alongside regular FDO counters in augmented FDO profiles. To use the initial example in Figure 1, since `foo()` contains hot calls to `bar()`, the files `a.c` and `b.c` would end up in the same cluster to enable cross module inlining later.

During the FDO optimization build, the compiler continues to compile one file at a time and reads in the augmented profiles. If a cluster has been formed in the step above, auxiliary source modules were specified in the profiles and are now read in and added to the compilation scope of the first, main module. This step sounds simpler than it actually is. This process can be conceptually thought of as combining multiple source files into one big one. Just as when doing this manually, problems with multiple definitions, identically named static variables, and type mismatches must be resolved by the compiler. To avoid redundant computation later, the compiler needs to keep track of what was specified in the main module.

Now the compiler has a greatly extended scope and intra-module inter-procedural optimizations can be performed. In our initial implementation we focused on the existing optimization passes inlining and indirect call promotion. Both passes had to be augmented to be able to handle functions from out of (original) scope.

After all transformations have been performed, unnecessary auxiliary functions are deleted to avoid redundant time spent in further optimization and code generation passes. Referencing the example in Figure 1 again, while compiling `a.c`, the compiler will read the auxiliary module `b.c`. After `bar()` has been inlined into `foo()`, the compiler can safely delete `bar()` from its current scope and not pass it on to later compilation phases. If it is referenced somewhere else, the body of `bar()` will still become available when module `b.c` is being compiled and linked in later.

The implementation of LIPO consists of three major blocks. Support for LIPO in the language frontend, where parsing of multiple modules must be supported, a runtime component, and compiler extensions to support optimizations. We will discuss details of these in the next sections.

3.1 Language Front-End Support

The main task for the language front-end is to support parsing of multiple source modules. This requires more than concatenating all source modules together, e.g., via include directives, and parsing the combined file. For languages like C++ the name lookup rules are very complicated and simply combining all sources and treating them as one extended translation unit won't work. For simpler languages such as C, gcc offers support with its `-combine` option, yet, even though this option has been implemented years ago, it is fragile, unrobust, and generally unused.

Our solution is to parse each module in isolation, i.e., we added support in the front-end to allow independent parsing of source modules. For gcc, this required clearing of the name bindings for global entities after parsing of each module. We shifted type unification and symbol resolution to the backend, which greatly simplified the required implementation overhead in the front-ends. Compilers with separated front- and back-end, e.g., open64, will pass compiler IR around and LIPO can be added easily as a simple extension.

3.2 LIPO Runtime

At the end of the program execution in the FDO training phase, before profiles are written, the IPA analysis takes place. For inlining we build the dynamic call graph. For indirect calls, we use the existing FDO value profiling to obtain the branch targets. For direct calls we don't rely on the existing edge profiles, but add new instrumentation for direct call profiling. The resulting counters are for consumption by LIPO only. This design sacrifices training phase execution speed in favor of smaller profile sizes, a decision we may revisit in the future.

Source module affinity analysis is now performed on this dynamic call graph. To obtain best module groupings, e.g., for inlining, it would be appropriate to model the inlining heuristics in the clustering algorithm. We, instead, use the simple greedy algorithm outlined in Figure 4. The computed module groups are written into the augmented profiles.

There is an interesting observation in regards to summaries. In traditional IPA, summary information is typically written to the IR object files in some encoding, and the link-time IPO has to read, decode and store this information in its internal data structures. Since LIPO runs at program runtime, summary information can be stored as program data and be used directly by LIPO, without the need for further decoding.

3.3 Optimization Extensions

There are several LIPO specific tasks in the compiler middle-end and back-end to enable cross module optimizations, to ensure correct code generation, and to reach a successful final program link.

3.3.1 In-core Linking

An in-core linking phase merges global functions, variables, and types across modules. Undefined symbols are resolved to their definition, if one exists in a module group. If function references can be resolved before the call graph is built, the implementation may chose to directly resolve calls in the intermediate representation.

As for the traditional link time IPO, type merging is also needed for LIPO. In this process, types from different modules are merged into a global type equivalence table, which is used by the compiler middle-end and back-end for tasks like type based aliasing queries, or useless type cast removal.

3.3.2 Handling Functions with Special Linkage

Functions defined in auxiliary modules have special linkage. Most of the functions are treated as inline functions, they are not not needed for code expansion and can be deleted after the final inline phase. COMDAT functions need to be expanded if they are still reachable after inlining. There may be multiple copies of a COMDAT function in a LIPO compilation. The compiler will pick one instance and discard the rest, similar to what the linker would have done in a regular link. Compiler-created functions, e.g., function clones after constant propagation, can never be 'external'. They can be deleted only if there is no remaining reference after inlining.

3.3.3 Static Promotion and Global Externalization

A *static* entity has internal linkage and a name that is either file- or function-scoped. Static variables and static functions need special handling in both main and auxiliary modules. Global variables need special handling in auxiliary modules as well. We distinguish these cases:

- For any module that is imported as an auxiliary module, static variables and functions defined in it need to be promoted to globals, both when the module is compiled as the main module and as an auxiliary module. The problem is that a unique, non-conflicting linker id for the static entities must be created that both caller and callee module can agree upon. Our solution is to postfix the original, mangled names with the keyword "lipo/cmo" and the main module's linker id. It is possible for multiple static variables in different scopes to share the same name. We therefore add sequence numbers to the names, following the variables' declaration order.
- When a module is never imported, no promotion needs to happen when that module is compiled as the primary module.
- Static functions in auxiliary modules are externalized, same as variables, but their function bodies are kept for the purpose of inlining. The naming convention is similar to the one for static variables described above.
- Global variables in auxiliary modules should be treated as, and converted to, extern.

3.4 Build System Integration

In this section we discuss integrating LIPO into build systems. We distinguish the three cases of a local fresh build, a local, but incremental build, and a distributed build.

For fresh builds on a local system, LIPO works without problems and, e.g., Makefile-based systems do not have to change. All sources are available, LIPO will find all auxiliary modules, and the dependence checking at the main module level (as opposed to including the auxiliary modules) is sufficient.

For incremental builds, the situation is slightly different. If a main module is part of a group and auxiliary modules are brought in during compilation, a modification of an auxiliary module makes a recompilation of the main module necessary. In order to maintain these dependencies, we developed a small tool that reads module profile information and dumps the full list of auxiliary dependencies. A common paradigm in Makefile based systems is to generate source dependencies. This tool can be added to this process. Dependencies should be refreshed whenever the FDO profiles are being regenerated, as modified profiles can lead to modified grouping decisions.

Integrating LIPO into a distributed build system, e.g., distcc [12], poses a similar problem. For such systems, the main module and all auxiliary files, headers, and profiles, need to be distributed across build machines. We can use the same tool introduced before and only minor modifications to these build systems are necessary to allow distributed LIPO builds.

4. Experimental Evaluation

In this section we analyze various aspects of our infrastructure, such as module grouping properties, IPA analysis overhead, compile time and file size overheads, as well as runtime performance. Most experiments were run on SPEC 2006 INT, using a cutoff ratio of 95% in our greedy clustering algorithm. We also present performance numbers for SPEC2000 INT, as well as interesting analysis results for a set of large industrial applications. For all these sets of benchmarks, we only compiled the C/C++ benchmarks, and only formed non-mixed language module groups. All experiments were run on an 8-core Intel Core-2 Duo, running at 2.33 GHz, with 32 GB of memory.

4.1 Module Groups

We analyze how the cutoff threshold in the greedy algorithm influences the module grouping. We compile all of SPEC2006 INT with cutoff thresholds of 80%, 90%, and 95%. The results are summarized in Table 2. For each experiment, we compute the total number of auxiliary modules used (column *Aux*), the number of trivial module groups, which are groups with only 1 module (column *Trv*), the maximum size of a module group for a benchmark (column *Mx*), and the average size of module groups (column *Avg*). To put these

1. Compute the sum total of all dynamic call edge counts.
2. Create an array of sorted edges in descending order of their call counts.
3. Find the cutoff call count:
   (a) Iterate through the sorted edge array and add up the call counts.
   (b) If the current total count reaches 95% of the count computed at step 1, stop. The cutoff edge count is the count of the current edge. A edge is considered *hot* if its count is greater than the cutoff count.
4. Start module grouping: For each call graph node, find all nodes that are reachable from it in a reduced call graph, which contains only hot edges. Add the defining modules of the reachable nodes to the module group of the node being processed.

Figure 4. Greedy clustering algorithm for module group formation.

numbers into context, we also provide the total number of modules for each benchmark (column *Mods*).

The experiments show that the average size of module groups is small, averaging (geometric mean) 1.3 for a cutoff of 80%, 1.5 for 90%, and 1.6 for 95%. The notable exception is the benchmark 403.gcc, which has oversized module groups because of remarkably flat profiles. Clearly, the graph partitioning algorithm has to improve in order to handle such cases properly. Another interesting benchmark is 473.astar, which only forms trivial program groups. Discounting 403.gcc, the largest module group consists of 11 files (in 483.xalancbmk at 95%).

4.2 IPA Analysis Overhead

LIPO's runtime overhead on the training phase consists of two components, the overhead from additional instrumentation to count direct calls, and the overhead from running the IPA analysis at the end of execution. The instrumentation runtime overhead is shown in Table 3, it amounts to 32.5% on average, with one case increasing by 2.6x. This overhead is a result of our design decision to not rely on basic block counts for direct call profiling, but to add more instrumentation instead.

We determine the IPA analysis overhead by measuring the difference in execution time of the instrumented binary, with and without the IPA analysis. This overhead is generally minimal. We show the runtimes without IPA in the *No IPA* column and overheads (comparing *LIPO* against *NoIPA*) larger than 1% in the following *Ov.* column.

4.3 Compile Time Overhead

We measured the overhead for full and incremental builds for SPEC2006 INT. For full rebuilds, we compare in Table 4 a regular build at -O2, a regular full FDO optimization build, and a full LIPO build. Since with FDO and LIPO there is generally more aggressive inlining, we would expect the FDO times to be higher than the -O2 times. The average group size at 95% cutoff was 1.6, which would lead us to expect a 60% compile overhead over FDO. However, we see a lower overhead of only 28%, as we are removing unnecessary code before it enters further optimization and code generation passes. Assuming unlimited parallelism in a distributed build infrastructure, a lower bound for full build times on such systems is the longest time it takes to compile any given module group. Since module groups are generally small, distributed build times can be a fraction of undistributed build times.

To measure the overhead for incremental builds, we run N experiments for each benchmarks, where N is the number of C/C++ files in this benchmark. In each experiment, we touch one source module, find all other groups containing this source file, and additionally touch each of these groups' main module, modeling the full dependencies in a build system. We then compute maximum and average rebuild overhead at parallelism levels 1 and 2 (indicated by letter 'j'). The results are also in Table 4.

The incremental build times are on average surprisingly low, less than 2 seconds on average, less than 20 seconds on average for the worst cases for sequential rebuilds, and less than 12 seconds on average for the worst cases at parallelism level 2. This number continues to decline at higher levels of build parallelism. The expected outlier is again 403.gcc, where too many large module groups were formed.

The columns *imax* and *iavg* further analyze the inclusion patterns and show to what extent modules are part of other module groups. We show maximum and average numbers, which help explain the incremental compile time overhead. For example, for 403.gcc, modifying a particular (unfortunate) file may cause 33 other files to be recompiled, roughly 22% of this benchmark's overall modules. We want to emphasize that even this pathological case is still significantly better than previous approaches, as outlined in Section 2, where all modules have to be rebuilt after IPO, even after insignificant code changes.

4.4 File Size Overhead

We analyze the changes in object file sizes and profile sizes for LIPO compared to a standard FDO build. In Table 5 we list these values for the SPEC2006 INT benchmarks. Column *objects* lists the sizes of the object files produced by the FDO-optimize build. The gcc compiler stores profile information in *gcda* files, and correspondingly, the column *profiles* lists the profile sizes. The following column shows the relation of these two in percent. The same data is shown for the LIPO object files, LIPO profiles, and the corresponding percentages.

Profiles add up to about 18% of the object file sizes for FDO. The file sizes for the LIPO objects increase by about 8% due to more aggressive inlining. The relative profile sizes for LIPO amount to 36% of the LIPO object files, as IPA information has been added to them. The LIPO profiles are about 2.1x larger than the FDO profiles. In total, LIPO imposes a total increase in file sizes of only 25% over the FDO baseline.

4.5 Performance

We analyze how the grouping cutoff threshold affects performance. In this paper we didn't add any novel optimizations, all gains come from more aggressive inlining and additional indirect call promotion.

For the cutoff values of 80%, 90%, and 95%, on the C/C++ SPEC2006 INT programs, we show the improvements of LIPO over the FDO baseline in Figure 5. For this particular comparison,

Clustering Cutoff:		80%				90%				95%			
Benchmark	Mods	Aux	Trv	Mx	Avg	Aux	Trv	Mx	Avg	Aux	Trv	Mx	Avg
400.perlbench	50	18	41	5	1.4	28	38	5	1.6	38	34	6	1.7
401.bzip2	7	3	4	2	1.4	4	4	3	1.6	4	4	3	1.4
403.gcc	143	216	82	13	2.5	365	65	18	3.6	524	54	26	4.4
429.mcf	11	1	10	2	1.1	3	9	3	1.3	3	9	3	1.3
445.gobmk	62	11	54	4	1.2	16	50	4	1.3	21	47	5	1.3
456.hmmer	56	1	55	2	1.0	1	55	2	1.0	1	55	2	1.0
458.sjeng	19	12	14	9	1.6	12	14	9	1.6	12	14	9	1.6
462.libquantum	16	1	15	2	1.1	1	15	2	1.1	1	15	2	1.1
464.h264ref	42	1	41	2	1.0	3	39	2	1.1	5	37	2	1.1
471.omnetpp	83	48	66	7	1.6	63	62	8	1.8	73	60	10	1.9
473.astar	11	0	11	1	1.0	0	11	1	1.0	0	11	1	1.0
483.xalancbmk	693	97	660	9	1.1	133	657	11	1.2	147	654	11	1.2
Geo Mean					1.3				1.5				1.6

Table 2. Module grouping information for SPEC2006 at 80%, 90%, and 95%

Benchmark	FDO	LIPO	Ov.	NoIPA	Ov.
400.perlbench	41.5	66.5	60.3%	64.3	3.4%
401.bzip2	70.3	75.1	6.8%	74.8	$< 1\%$
403.gcc	2.0	2.6	29.2%	2.6	$< 1\%$
429.mcf	41.5	40.4	-2.7%	40.5	$< 1\%$
445.gobmk	173.9	208.0	19.6%	207.2	$< 1\%$
456.hmmer	106.5	106.9	0.4%	107.5	$< 1\%$
458.sjeng	236.3	290.3	22.9%	290.9	$< 1\%$
462.libquantum	2.8	2.9	5.4%	2.9	$< 1\%$
464.h264ref	148.5	241.1	62.3%	241.0	$< 1\%$
471.omnetpp	110.8	209.6	91.3%	208.3	$< 1\%$
473.astar	150.1	161.9	7.9%	160.7	$< 1\%$
483.xalancbmk	175.8	456.8	159.8%	447.4	2.1%
GeoMean			32.5%		

Table 3. Training phase runtimes, in $[sec]$, and overhead from additional instrumentation, and IPA.

we also add the performance numbers generated with a cutoff of 99%. All experiments were compiled with -O2.

We see several degradations at a cutoff threshold of 99% over 95%. Clearly, the compiler can benefit from better tuning for larger compilation scopes. For now, we picked a default threshold of 95% in our implementation. As other IP optimization heuristics improve, this value will be subject of further tuning. The threshold of 95% also results in good performance results on SPEC2000 INT, shown in Figure 6, which yields an overall performance improvement of about 4.4%..

4.6 Large Applications

We verified – and confirmed – the presented results on a set of larger, industrial C++ applications running in Google's datacenters. In Table 6 we present the performance numbers for seven large and two smaller benchmarks. Average performance on these benchmarks improves by 2.9% (over the FDO baseline). In this table we also show the module grouping information, similar to Table 2.

The average module groups sizes are, again, surprisingly small at 1.24. To find out why we show the total number of dynamic call graph edges, their sum total edge counts, and the number and percentage of hot call edges, as filtered out by our greedy algorithm (Table 4). We find that on average only about 5% of all edges are *hot* (3.6% if we discard the smaller Video Converter benchmark).

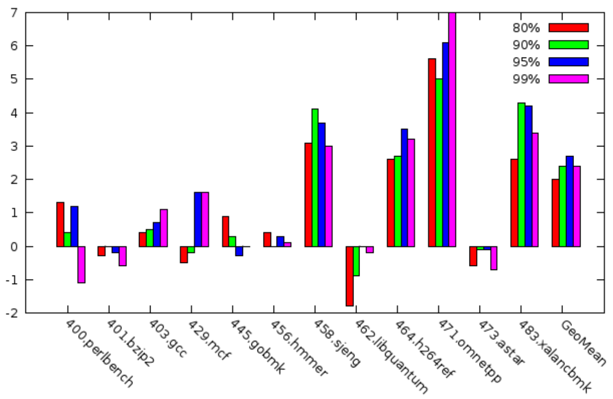

Figure 5. Performance improvements in [%] over FDO, for SPEC2006 INT, at cutoff thresholds of 80% (2.0%), 90% (2.4%), 95% (2.7%), and 99% (2.4%)

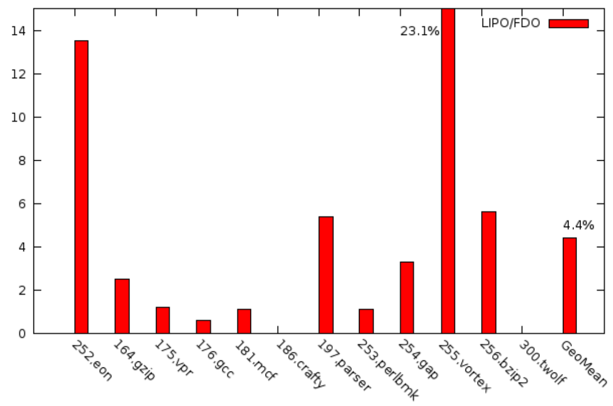

Figure 6. Performance improvements in [%] over FDO, for SPEC2000 INT

	Full Builds, j1				Incremental Builds					
Benchmark	-O2	FDO	LIPO	Overhead	j1 max	j1 avg	j2 max	j2 avg	imax	iavg
400.perlbench	42.3	43.9	56.7	29.0%	29.3	3.1	16.3	2.1	12	1.7
401.bzip2	3.0	4.6	5.9	41.7%	2.5	1.1	1.9	1.0	3	1.4
403.gcc	108.0	129.0	255.0	97.7%	126.1	7.7	68.1	4.8	33	4.4
429.mcf	1.0	1.6	2.6	60.0%	1.6	0.4	1.1	0.3	2	1.3
445.gobmk	30.4	35.4	41.0	15.7%	16.6	1.4	8.7	1.1	9	1.3
456.hmmer	12.0	11.3	11.5	1.3%	1.1	0.3	1.1	0.3	2	1.0
458.sjeng	4.4	5.1	6.6	30.0%	3.1	1,0	2.2	0.8	4	1.6
462.libquantum	1.8	1.8	1.8	0.0%	0.6	0.1	0.5	0.2	2	1.1
464.h264ref	20.6	26.7	27.7	3.7%	7.2	1.1	5.4	1.0	3	1.1
471.omnetpp	39.5	38.8	63.6	64.0%	18.2	3.1	10.6	2.0	7	1.9
473.astar	2.0	2.6	2.6	-0.4%	0.5	0.3	0.5	0.3	1	1.0
483.xalancbmk	278.7	257.0	333.5	29.8%	63.3	2.5	35.3	2.0	16	1.2
Geo Mean				28.0%	18.8	1.9	11.5	1.3		1.6

Table 4. Full and incremental rebuild times for SPEC 2006 INT, in $[sec]$, at parallelism level 1, 2

	FDO			LIPO		
Benchmark	objects	profiles	%	objects	profiles	%
400.perlbench	2406827	491960	20.4%	2660211	876172	33.0%
401.bzip2	113082	18028	15.9%	141074	33852	24.0%
403.gcc	8145095	1450064	17.8%	10081375	3036424	30.1%
429.mcf	49456	4436	8.0%	63176	17048	27.0%
445.gobmk	5631666	353504	6.3%	5845274	632056	10.8%
456.hmmer	560264	101948	18.2%	561944	225048	40.1%
458.sjeng	341184	39020	11.4%	372416	83156	22.3%
462.libquantum	86424	13800	16.0%	86408	40272	46.6%
464.h264ref	1098496	136244	12.4%	1094216	261684	23.9%
471.omnetpp	2160915	423876	19.6%	2585539	1053220	40.7%
473.astar	98920	17436	17.6%	98272	51892	52.8%
483.xalancbmk	17856968	3975728	22.3%	18399040	8669508	47.1%
Total	38549297	7026044	18%	41988945	14980332	36%
Overhead	45575341			56969277		*25%*

Table 5. Object and profile file sizes in [byte], and overhead, for SPEC2006 INT, for FDO and LIPO

We attribute the small average group size to the fact that this C++ source base has been tuned for years, and, in absence of an inter-procedural optimizer, all performance critical code has been moved into C++ header files.

5. Discussion

In this paper we made a case for combining inter-procedural optimization with FDO and describe and contrast existing IPO frameworks with our new LIPO approach. Compared to existing approaches, we no longer require writing of the compiler IR to disk, no longer need a link time optimizer, and minimize code regeneration overhead. This design leads to improvements of an order of magnitude for compile time and resource requirement and makes IPO amenable to distributed build systems.

Training phase runtime overhead is 32% on average. We made an explicit design decision to trade in this runtime penalty against small profile file sizes, assuming that training runs are generally short. This is a decision we may want to revisit in the future, or offer under an option.

The file size overhead is at 25%. This is an order of magnitude smaller than the 4x to 10x overheads of existing infrastructures. The compile time overhead for full builds is less than 30%, compared to factors of 2x and higher for existing infrastructures, and builds can be distributed, which is not easily possible in existing systems. Incremental builds are done in seconds on average, compared to minutes and hours in existing approaches, which always have to run a full IPO and a full code generation phase, even after small code changes. This fast behavior is enabled by small average module groups containing only 1.3 - 1.6 files, and deletion of redundant compiler IR.

LIPO's current main disadvantages are the following.

- We currently don't support full program analysis. While of course all program modules can be grouped into a single compilation, e.g., similar to LLVM's approach, such an approach won't scale to very large applications.
- LIPO does require FDO.
- Mixed language support is hard, in particular for compilers that maintain language-specific front-ends. In this paper, we only generated non-mixed language module groups.

The results presented in this paper are based on gcc 4.4. An updated implementation for gcc 4.5 is available on the public gcc branch *lw-ipo*. We believe that the concepts in LIPO are general and can be implemented in other compilers.

Benchmark	Perf	Mods	Aux	Triv	Mx	Avg	CG Edg	Sum Total	Hot	%
BigTable	-0.73%	1803	1519	1613	63	1.84	52576	2077481070	4223	8.0%
Ad Delivery	+4.56%	2441	329	2365	18	1.13	28014	8660809169	934	3.3%
Indexer	+0.54%	3631	514	3471	18	1.14	56937	1325046290	1594	2.8%
Search	+1.48%	2410	369	2323	20	1.15	28748	3067176774	1003	3.5%
OCR	+3.61%	1029	322	952	20	1.31	7226	290777838	476	6.6%
Search Quality	+3.19%	1232	36	1218	10	1.02	15540	3272632527	95	0.6%
Sawzall	+0.62%	2285	103	2254	15	1.04	30072	435312950	227	0.7%
Video Converter	+6.52%	86	39	78	13	1.45	1797	358857930	306	17.0%
Compression	+6.99%	224	15	218	7	1.06	2271	126211312	81	3.6%
Geo Mean	+2.94%					1.24				5.0%

Table 6. Industrial applications performance, module grouping information, and statistics on dynamic call graph edges, sum total of all edge weights, and number/percentage of hot edges.

6. Future Work

Besides fine tuning of the clustering algorithm and relevant heuristics, we will gradually add existing IP optimizations to LIPO, many of which have been mentioned in the introduction. We plan to work on some of LIPO's disadvantages, in particular, we are evaluating summary based approaches to allow classic IPO optimizations that need whole program analysis. We're also interested in using sample based profiling instead of instrumented FDO, as this would eliminate the FDO instrumentation and training phase [17]. The fact that the IPA analysis phase runs at execution time allows for the development of interesting new techniques, which we are evaluating. This should become a rich area of research.

7. Acknowledgements

We wanted to thank Urs Hoelzle for his early comments on this paper, as well as the anonymous reviewers for their invaluable feedback.

References

[1] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *ECCOP '96: Proceedings of the 10th European Conference on Object-Oriented Programming*, pages 142–166, London, UK, 1996. Springer-Verlag. ISBN 3-540-61439-7.

[2] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. *SIGPLAN Not.*, 32(5):134–145, 1997. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/258916.258928.

[3] Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable cross-module optimization. *SIGPLAN Not.*, 33(5):301–312, 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/277652.277745.

[4] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 59–70, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: http://doi.acm.org/10.1145/143165.143180.

[5] Dhruva R. Chakrabarti, Luis A. Lozano, Xinliang D. Li, Robert Hundt, and Shin-Ming Liu. Scalable high performance cross-module inlining. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 165–176, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2229-7. doi: http://dx.doi.org/10.1109/PACT.2004.25.

[6] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling C programs. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 246–257, New York, NY, USA, 1989. ACM. ISBN 0-89791-306-X. doi: http://doi.acm.org/10.1145/73141.74840.

[7] Chris Lattner, Private Communication.

[8] Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 58–67, New York, NY, USA, 1986. ACM. ISBN 0-89791-197-0. doi: http://doi.acm.org/10.1145/12276.13317.

[9] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Softw. Pract. Exper.*, 21(6):581–601, 1991. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.4380210604.

[10] Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Softw. Pract. Exper.*, 18(8):775–790, 1988. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.4380180805.

[11] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, London, UK, 1995. Springer-Verlag. ISBN 3-540-60160-0.

[12] distcc, A fast free distributed C/C++ compiler. http://distcc.samba.org. URL `http://distcc.samba.org`.

[13] M.W. Hall. Managing interprocedural optimization. In *PhD Dissertation*, 1991.

[14] Donald E. Knuth. An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2): 105–133, 1971. doi: 10.1002/spe.4380010203. URL `http://dx.doi.org/10.1002/spe.4380010203`.

[15] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO04)*, March 2004.

[16] Sungdo Moon, Xinliang D. Li, Robert Hundt, Dhruva R. Chakrabarti, Luis A. Lozano, Uma Srinivasan, and Shin-Ming Liu. SYZYGY - a framework for scalable cross-module IPO. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 65, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

[17] Vinodha Ramasamy, Paul Yuan, Dehao Chen, and Robert Hundt. Feedback-directed optimizations in gcc with estimated edge profiles from hardware event sampling. In *Proceedings of GCC Summit 2008*, 2008.

[18] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. In *Journal of Programming Language*, pages 1–18, 1992.

[19] The Open64 Compiler Suite. www.open64.net. URL `http://www.open64.net`.