Reprinted from the

Proceedings of the GCC Developers' Summit

July 18th–20th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Ian Lance Taylor, *Google*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Interprocedural optimization framework in GCC

Jan Hubička
SuSE ČR, s. r. o.
jh@suse.cz

Abstract

Ongoing effort to implement GCC framework for interprocedural optimizations is described with special focus on recent reorganization to work on Single Static Assignment (SSA) form. The paper also give high level overview of existing and planned interfaces and optimization passes.

1 Introduction

GCC was originally designed to optimize on the scope of basic blocks (local optimizations), later extended to whole function analysis using the SSA form (global intraprocedural optimizations, [13]). Optimizing across function and compilation unit boundaries (interprocedurally) is the next step.

In this paper we describe effort to extend Tree-SSA framework to interprocedural scope. The current state is result of several projects, in particular the reorganization of GCC backend to new intermediate language (GIMPLE) [13], reorganization of compilation process to be driven by callgraph module [7], and several projects to implement or improve interprocedural optimizations [3, 11, 12, 10, 8] and analysis [5, 6].

2 Basic interfaces

One of main design goals is to make writing of interprocedural passes similar to writing intraprocedural optimization passes using Tree-SSA framework [13]. We outline the main interfaces:

2.1 Pass manager

The compilation is driven by a pass manager able to manage both kinds of passes. The pass queue is a nested structure allowing to define a sub-passes of every pass. At the moment, all top-level passes are executed interprocedurally (called just once for whole unit), the subpasses are intraprocedural and are called for each function separately in the topological order across callgraph.

Pass-manager is also able to execute simple function cleanups after or before pass, so called TODOs. The cleanups are executed on current function for intraprocedural passes, or for all functions for interprocedural passes.

2.2 Cgraph and Varpool

The compiled unit is represented using callgraph and varpool [7]. Callgraph is a directed multigraph whose nodes are functions and edges call sites. Varpool is a list of static (public, private and function local) variables in the compilation unit. Both datastructures can be walked via several `FOR_EACH_*` constructs defined in `cgraph.h`, several utilities to analyze the datastructures are present in `ipa.c`, `ipa-prop.c` and `ipa-utils.c`.

The varpool and cgraph nodes allows optimization passes to store information about variables and functions either directly into the structures or indirectly via on-side arrays indexed by IDs. The second choice is preferred for all data that are not required to live across whole compilation to avoid liverange problems mentioned in Section 6.

The information stored in cgraph nodes is divided into three sections — local data are results of function wide analysis, global data are results of interprocedural propagation of local data and RTL data are data used during RTL expansion (for purposes such as computing required stack alignment). `cgraph_local_info`, `cgraph_global_info`, `cgraph_rtl_info` accessors verify that the data is available and in future will allow to allocate the data only for those callgraph nodes, where needed, if callgraph starts to consume considerable amount of memory.

2.3 Other datastructures

The bodies of functions are represented in GIMPLE intermediate language with control flow graph (CFG), single static assignment form (SSA) and profile information available after the passes constructing them was executed.

At the time of writing this paper, majority of functions to operate with GIMPLE was not updated for interprocedural optimization and require variables `current_function_decl` and `cfun` to be both set to function to operate on. There is plan to add explicit function argument replacing `cfun` references to all basic GIMPLE manipulation function and also add `GIMPLE_` prefixes at the same time. In meantime, `push_cfun` and `pop_cfun` functions are available for current function manipulation.

It is important to keep in a mind that not all datastructures are localized for each function, in particular the dominator tree, SSA updating information and alias analysis are hold in global variables. This conserve memory usage, but also require a care to be taken to not keep the datastructures alive after changing function context. A short term plan is to add `push_function` and `pop_function` responsible for switching `cfun` and `current_function_decl` and invalidating the datastructures not supposed to be carried over to a different function.

Also GCC implementation of bitmap datastructure uses single memory storage to hold various bitmaps (a default bitmap obstacks). This storage is recycled after each function compilation, so it is important to not use the default bitmap obstack in interprocedural passes and data supposed to survive across function boundary.

3 Scope of interprocedural optimization

Interprocedural optimizations can be implemented with different scopes. The larger portion of the program is seen by optimizers, the greater effectivity of the optimization is. In GCC we define 4 main scopes.

3.1 Function at a time

Most memory conservative compilation is to output into assembly file functions as they are parsed from source code. This allows just very basic interprocedural optimization propagating information in the source code order. For example simple forward inlining pass is implemented by keeping in memory all sufficiently small inlinable functions (inline candidates) and inlining them into a callers that gets compiled later in the process.

This mode of compilation was used by GCC until version 3.4 (when callgraph project was merged [7]) and it is still available via `-fno-unit-at-a-time` command line option. GCC, when optimizing, is usually not using this scheme anymore. Precise setting is language specific: C++ never use function at a time, C, Ada and Fortran is function at a time at `-O0` optimization level, Java for all optimization levels by default. For Java, the default was changed because of memory usage problems: it is common to build application from bytecode library and thus load into compiler large units at once that cause large memory consumption otherwise. This problem is more discussed in Section 6.

Function inlining and simple forward propagation during RTL expansion are the only interprocedural optimizations enabled at this level.

3.2 Unit at a time

In unit at a time compilation the whole compilation unit is parsed first and interprocedural optimizations then operate on complete callgraph. The compilation unit does not necessarily need to correspond to source level unit. C frontend implements `--combine` command line option [9] allowing to compile multiple source units at once. Java frontend is able to compile multiple units from bytecode library, for Fortran simple concatenating of source files works well enough. Multiple source units compilation is not available in particular for C++, where implementation is planned for multiple reasons including the reduced need for template specializations and thus overall speedup and reduction of disk space needed to build larger C++ project. KDE project already adoped a scheme, where sources from each directory are included together in order to reach similar benefits.

3.3 Link time optimization

Link time optimizers replace regular object files by object files containing intermediate language. The linker

then calls back to compiler to compile whole application or library at once. This scheme has significant advantages over compilation of multiple source files into single object file (as with `--combine`) especially by being transparent to build system and not requiring user to modify makefiles. With link time optimization also multiple languages can be compiled together and the debug cycle (modify—rebuild operation) is faster.

Link time optimization is one of major components missing in GCC and is currently available only for Java bytecode. Work to implement frontend independent link time optimizer is being done on `lto-branch` [1].

3.4 Whole program optimization

For many interprocedural optimizations it is important to know more about use of public functions and variables defined in the compilation unit than is made explicit by common programming languages.

In whole program mode all functions and variables defined in the compilation unit, except for `main`, are considered static. This mode still allows calling into external library. Additional entry points can be added via `externally_visible` attribute.

This give explicit information about what public variables and functions in the compilation unit are used by compilation unit itself and what can be referenced from libraries too. This compilation mode is available via `-fwhole-program` command line option. Until linktime optimization gets implemented, it is useful only in combination with `--combine`, otherwise by properly declaring variables and functions static has precisely the same effect.

4 Pass queue organization

The basic structure of pass queue at `-O2` optimization level is as shown in Figure 1. Individual interprocedural passes are separated by ↓. Local passes are always executed on the callgraph in topological order to allow easy propagation of information from callees to callers.

4.1 Parsing stage

During the parsing, frontend is responsible for building functions and variables. Functions are passed in

Parsing:
Parsing
Registering functions/variables in the callgraph
↓
End of source level unit:
cgraph construction
↓
unreachable function removal
End of unit:
↓
function visibility
↓
unreachable function removal
SSA conversion
Early inlining
Early optimization
↓
Inline plan decision
↓
Referenced variables
↓
Pure/const discovery
↓
Inlining
Alias analysis
GIMPLE local optimizations
RTL local optimizations

Figure 1: The pass queue

GIMPLE format, variables initializers are in frontend specific trees. Once construction is done `cgraph_finalize_function` or `varpool_finalize_decl` is called and the functions and variables are registered to middle end. From that point it is middle end responsibility to decide on when the functions and variables are going to be compiled and released from memory.

At the end of each source level compilation unit, `cgraph_finalize_unit` is called that is responsible for lowering the functions into low GIMPLE (with control flow graph built), producing a callgraph and finally eliminating unreachable functions. The analysis are done on demand, so only reachable functions are lowered to reduce memory usage. `cgraph_finalize_unit` can be invoked several times by frontend without re-processing already lowered functions (in case a multiple source level compilation units are being compiled) so the unreferenced variables and

functions are released from memory during the parsing process.

Lowering passes are controlled by the pass manager and the toplevel queue is `all_lowering_passes`.

Once frontend is done with parsing, `cgraph_optimize` is called and rest of compilation process is driven by the middle end. In next subsections describe passes in `all_ipa_passes` executed from `cgraph_optimize`.

4.2 Early optimizations

Early optimization consists of several intraprocedural subpasses executed on each function in sequence. The functions are ordered in topological order of the callgraph so information can be easily propagated from callees to callers.

Further lowering is done here: profile instrumentation is inserted, simple CFG cleanups applied, OpenMP constructs are expanded and single static assignment (SSA) form is build.

Important part of early cleanups is early inlining pass. This simple pass is inlining functions already processed by early optimizations that are smaller than the expected function call overhead. (Size of the whole compilation unit thus monotonically reduce after each inlining caused by early inliner.) Implementation of early inliner is shared with implementation of inliner for function at a time mode. Effect of early inlining to performance of C++ benchmark TraMP-3d is shown at Figures 2 and 3. Similar effects was measured on several other C++ benchmarks too.

Early optimization pass is very effective for many C++ programs. Most wrapper functions and simple accessors are inlined into the function body enabling intraprocedural early optimization to be effective. Without early inlining, most functions in C++ would actually be just simple accessors or contain only calls to other functions with no oppurtunity for intraprocedural optimization. Early inlining is however not able to resolve properly code size growth and performance tradeoffs, so less trivial decisions needs to be done by a full inlining pass.

Simple SSA optimization passes are applied next: constant propagation, forward propagation, scalar replacement, copy propagation, Φ merging, dead code removal,

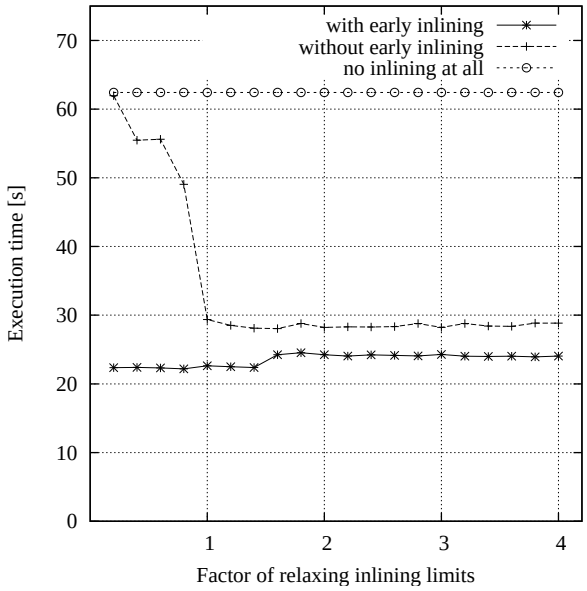

Figure 2: Effect of relaxing inline limits on performance of TraMP-3d. Chart courtesy of Martin Jambor [8].

tail recursion pass and profile estimation. These passes are regularizing function bodies so simplistic analysis done by interprocedural passes (in particular cost estimates in inliner) are more realistic and less dependent on particular coding style.

At the moment, no attempts are made to use alias analysis information and perform optimizations on memory references. Experiments has however shown that basic transformations, like dead store removal, are very important. Simple dead store removal pass looking for write only local variables is ablto to remove during compilation of TraMP-3d benchmark over 1000 variables. This results in important compilation time improvements and allows to limit overall function growth parameter of inliner (from 60% to more reasonable 20%) potentially speeding up compile times by about 30% as well as reducing sizes of resulting binarries. Only known regression caused by introduction of early optimization pass, a 10× slowdown on one of FreeFEM3d benchmarks is eliminated by this transformation too. The slowdown is caused indirectly by driving inliner to produce a function body with number of accesses exceeding limits of alias analysis Removing the memory references early makes work of alias analysis a lot easier.

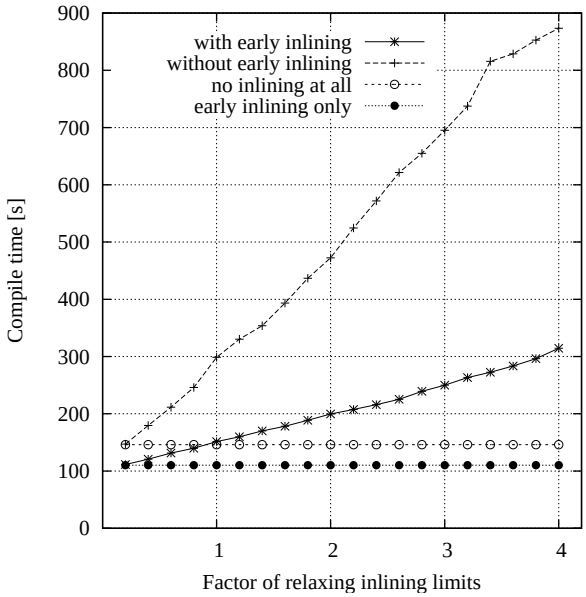

Figure 3: Effect of relaxing inlining limits on compile times of TraMP-3d. Chart courtesy of Martin Jambor [8].

4.3 Interprocedural passes

While early optimizations are done in topological order propagating from callees to callers, the interprocedural optimizations can access whole callgraph.

Interprocedural constant propagation [10] (with function cloning), inline plan decision [7], analysis of static variable references, pure and const function discovery, type escape and points to analysis is performed at this stage. Most of the passes are followed by a cleanup removing unreachable functions from the callgraph.

4.4 Intraprocedural optimization

Rest of compilation process is again organized as a sequence of intraprocedural passes executed in topological order on the callgraph. The alias information is built making SSA optimization passes more effective. All SSA optimization passes (including ones already performed in early optimizations) are executed, function bodies are lowered to machine specific RTL intermediate language, further optimized and output to assembly file.

While performing intraprocedural optimization, list of variables still referenced by the function bodies after optimization is collected and only referenced variables are output. No attempt is made to remove functions made unreachable by the late optimization, because the toplogogical order does not allow that (the functions rendered unreachable are already output into assembly), however the topological order allows to forward propagate some machine specific information, such as preferred stack alignment of the callee or calling conventions. The early optimization and interprocedural passes should be effective enough to render most unreachable functions dead early, so this limitation should not cause major problems. Additional problem is that resulting binary execute backwards (ie callees are placed in program before callers). It would be desirable to implement function reordering pass that use assembler subsections or fragments to reach the proper function ordering.

5 How link time optimization fits in

To implement link time optimization it is necessary to change representation of object files so they contain intermediate language rather than machine instructions and make linker to call back into compiler to optimize those "fake" object files.

Link time optimization is major missing component in the GCC interprocedural optimization framework. Work is being done on `lto-branch` to implement writing and reading of GIMPLE [1], in this section we just outline how the GIMPLE reader/writer will fit into existing framework.

Based on current optimization pass queue (Figure 1), writing of IL should happen just after early optimization, while after reading, early inlining should be done once again (enabling crossmodule inlining) followed by re-optimizing the functions where we inlined into followed with the rest of optimization queue unchanged.

In order to allow as much of parallelism as possible and to reduce modify—rebuild cycle, it is very desirable to push as much work as possible into the compilation stage. It might make sense to run a limited version of other interprocedural passes (in addition to early inlining) and include more passes in early optimizations as long as resulting object file size decrease noticeably. Since almost all work needs to be redone at crossmodule scope, pushing optimization passes up has limitations.

6 Scalability problems

GCC is latecomer into world of interprocedural optimizations. With source bases growing in their size it is important to not underestimate the scalability issues. The goal is to make the framework scalable enough to allow building all packages of common open source distribution with interprocedural optimization enabled. This has been partly reached with unit at a time compilation that hits important limitations only for Java programs. However for link time optimizations it is clear that the framework is not scalable enough.

6.1 Memory consumption

Memory consumption seems to be a major problem. To compile GCC backend with intermodule compilation, currently over 2GB of RAM (on 64bit machine) are needed. To deal with the problem, several steps was taken.

Per-line memory allocation statistics are available in compilers configured with `--enable-gather-detailed-mem-stats`. With `-fpre-ipa-mem-report` the memory usage just after parsing the whole compilation unit is reported, with `-fpost-ipa-mem-report` memory usage just after interprocedural passes is reported and `-fmem-report` reports memory usage at the end of compilation.

Memory usages on compiling C language testcase (`combine.c`) is shown in Figure 5. The numbers represent reachable garbage collected memory on two points of compilation (before and after interprocedural passes), not the overall memory allocation. As can be easily seen, the memory is quite evenly distributed in between various datastructures so in order to significantly reduce the memory usage many of them needs to be reengineered.

Goal of `tuples-branch` is to reorganize GIMPLE memory representation into flat form [4]. This will couple several datastructures into single structure. For usual binary operation this means replacing statement list (48 bytes), statement annotation (96 bytes), GIMPLE modify statement (48 bytes), binary expression (64 bytes) into flattened form of roughly 90 bytes (for operation with no memory operands). Somewhat overestimating the savings by assuming that all 20565 statements of `combine.c` are such a binary operations, `tuples-branch` should save about 3.4MB out of 21MB or 16%. On the other extreme assuming that all operands do have memory operands, one would get 2.5MB, 11%. For DLV testcase the relative memory reduction would be smaller because larger portion of memory is consumed by other structures.

Promising plan is eliminate labels on GIMPLE form and replace label references by direct basic block pointers. Memory consumption for 6237 artifical labels in `combine.c` together with associated datastructures (annotations, statement list, label expression) is 2MB, 10%. Further reductions are possible by simplifying memory representation of conditional gotos and switch that current duplicate information available in the control flow graph.

Important problem represents variable annotations. The consumption of annotations themselves is about 2%, however the data stored in annotations consist almost exclusively from datastructures local to optimization passes that are maintained live for whole program. The unused pointers from variable annotations point to another datastructures (such as dead statements) and blocks garbage collector from freeing the memory. It is difficult to get precise statistics on how many datastructures are kept alive by those dangling pointers, however in the past problems like variable annotations pointing to dead statement pointing to list of free SSA names pointing to all other dead statements defining SSA name was identified and fixed. In order to avoid such problems to appear, an infrastructure needs to be introduced. There are plans for adding `ggc_unreferenced` function that tells garbage collector that given datastructure should be dead and for compiler compiled with checking it would instruct garbage collector to report datastructures marked by this flag but still referenced via dangling pointers.

Further easy places to reduce overall footprint include reduction of SSA operand cache representation, control flow graph representation and eliminating need for majority of temporaries by producing naked SSA names not referencing to variable declarations. For C++ the elimination of dead scope blocks and duplicate declarations needed only for debugging should do a miracles (saving up to 50% of post-IPA memory footprint).

6.2 Not loading whole program at once

For link time optimization, the interprocedural optimizer can be organized in a way so all function body analysis are performed first and saved into object files as function summaries (including callgraph, function body size characteristics, jump functions for constant propagation etc.). The link time optimizer then can perform interprocedural optimizations exclusively using those summaries completely avoiding a need to load all function bodies into memory at once.

This scheme was partly implemented in earlier version of GCC interprocedural optimization framework [7], however it imposed restrictions on pass ordering and brought other implementation challenges. It was decided to put this plan on hold for a moment until the implementation matures enough to make clear what interprocedural passes shall be implemented and in what order.

Alternative approach is to introduce two intermediate languages—one incomplete but memory conservative to allow interprocedural optimization on and second more complete for local optimization.

6.3 Compilation time problems

In general interprocedural optimizations does not significandly increase compilation time, because analysis and transformations implemented are very simple. On SPEC2000 build, unit at a time compilation is 8% slower. The slowdown is mostly caused by inliner producing more intermediate code and slowing down the local optimization. Crossmodule compilation is 21% slower, this time extra cost of garbage collection is dominant. Whole program compilation 15% slower. Slowdown is smaller because more unreachable function elimination is performed and inline decisions are improved.

Major problem of link time optimization is however elimination of parallelization opportunities. While compilation can be easily parallelized by running multiple compilations at once, linking is serial. Partly the problem can be eliminated by moving as much work as possible into parsing as discussed in Section [1]. It is clear that at some point in future parallel linking will need to be developed. This is probably best done by performing interprocedural optimizations in single process and letting other nodes to perform local optimization on individual function bodies.

7 Benchmarks

Figure 7 shows performance of SPECint2000 on AMD Opteron machine. The benchmarks consist from hand tuned C programs, so interprocedural optimization is not really mandatory. Largest amount of speedup come from unit at a time compilation and inlining, crossmodule and whole program optimization adds just little extra gain. More speedups are possible by implementing more specialized interprocedural transformations (such as promoting datastructures to smaller type or replacing linked lists by arrays). Those transformations are however not believed to be as important for real world applications and thus are not high priority at this stage of development.

To benchmark interprocedural optimizations on more modern code, an C++ benchmark suite was put together. The benchmarks are run nightly and made available at `http://www.suse.de/~gcctest`. The benchmarks set up by Richard Guenther was chosen for their high abstraction penalty and heavy use of C++ constructs (mainly the templates). The more interesting benchmarks include

- The TraMP-3d hydrodynamics simulation code. This benchmark has extreme abstraction penalty caused by function call. There are several times more function calls eliminated by inlining that number of instructions executed in the resulting program.
- The wave C preprocessor that comes with the Boost library.
- The botan C++ library of cryptographic algorithms.
- A benchmark suite exercising the DLV, a Disjunctive Datalog System (and more).
- The FreeFEM3D finite element library benchmark collection.

Unlike CPU2000 benchmarks, both the source code and test inputes for the C++ benchmark suite is freely available with the exception of DLV.

Many of the testcases shows important improvements in runtime, compilation time, object size and compiler memory usage. Among the most important changes in compiler are believed to be:

- Early optimizations combined with inlining.
- Improvements to alias analysis representation and grouping heuristics.
- Refined inlining heuristics (in particular better size estimates).

8 Future plans

In addition to already mentioned `tuples-branch` [4] and `lto-branch` [1] there are number of planned extensions. Including:

- Overall API cleanups, in particular removal of dependency on global variable specifying current function in basic GIMPLE manipulation API.
- Inliner should be improved to be able to handle calls via callback and to support partial inlining.
- Framework for dataflow on individual fields of structures [8] and for context sensitive dataflow.
- Basic interprocedural pass to regularize function call conventions (ie replace values passed in structures by individual fields, remove dead parameters, and to convert parameters passed by reference to parameters passed by value.
- Alias analysis information available to early optimization and interprocedural passes (either in SSA form or via query system).
- More interprocedural passes (matrix flattening, structure reorganization and interprocedural points-to analysis is in development).
- The link time optimization task can be decomposed into number of subtasks including: developing GIMPLE writer and reader, developing type information writer and reader, seprating of frontend and backend (removing langhooks and frontend use of targhooks), developing type system able to represent multiple languages at once and reducing memory usage.
- Fortran currently produce wrong callgraph assigning each function multiple declarations. This prevents inlining and should be solved by a pass combining different declaration into one. Similarly C `--combine` is too strict on type copatibility so SPECint2000 can not be compiled without several hacks to the compiler.
- Java and Fortran both have potential of being interesting for testing interprocedural optimization framework.

References

[1] D. Berlin, D. Edelsohn, S. Ellcey, S.-M. Liu, T. Linthicum, M. Meissner, K. Zadeck, M. Mitchell: Link-Time Optimization in GCC: Requirements and High-Level Design

[2] D. Novillo: Design and Implementation of Tree SSA, GCC Summit Proceedings, 2004, p 119–130

[3] M. Hagog, C. Tice: Cache Aware Data Layout Reorganization Optimization in GCC, GCC Summit Proceedings, 2005, p 69–92

[4] R. Henderson, A. Hernandez, A. Macleod, D. Novillo: Tuple Representation for GIMPLE

[5] J. Hubička: Interprocedural optimization on function local SSA form in GCC, GCC Summit Proceedings, 2006

[6] J. Hubička: Profile driven optimisations in GCC, GCC Summit Proceedings, 2005, p 107–124

[7] J. Hubička: The GCC call graph module: a framework for inter-procedural optimization, GCC Summit Proceedings, 2004, p 65–78

[8] M. Jambor: Optimization in the GNU Compiler Collection Targeted at Scientific Computing, diploma thesis, 2007 (in preparation)

[9] G. Keating: Inter-module Analysis in GCC, GCC Summit Proceedings, 2005, p 125–132

[10] R. Ladelsky: Interprocedural Constant Propagation and Method Versioning in GCC, GCC Summit Proceedings, 2005

[11] R. Ladelsky: Matrix flattening and transposing in GCC, GCC Summit Proceedings, 2006

[12] M. Namolaru: Devirtualization in GCC, GCC Summit Proceedings, 2006

[13] D. Novillo: Design and Implementation of Tree SSA, GCC Summit Proceedings, 2004, p 119–130

Runtime and binary size

benchmark	ICC		GCC 4.1		mainline	
Tramp3d	336.82s		19.65s	2025Kb	13.54s	1819Kb
Boost	4.10s	2734Kb	2.12s	1070Kb	2.36s	810Kb
Botan (geomean)	44.39MB/s		46.66MB/s	2006Kb	48.04MB/s	1778Kb
DLV (geomean)	6.48s		5.34s	373Kb	5.6s	320Kb

Build times and compiler memory usage

benchmark	ICC	GCC 4.1		mainline	
Tramp3d	74.38s	113.82s	1291Kb	79.74s	1819Kb
Boost	216.05s	180.08s	1076Kb	112.15s	810Kb
Botan		174.34s	167Kb	162.71s	121Kb
DLV		204.99s	373Kb	112.00s	320Kb

Figure 4: Summary of C++ benchmark results

	pre-IPA bytes	rel	post-IPA bytes	rel
statements:				
trees in statements	4282472	23.65%	3408408	15.63%
stmt annotations	2168544	11.97%	1933152	8.86%
stmt lists	1137272	6.28%	1353936	6.21%
declarations:				
labels	959800	5.30%	1008064	4.62%
temporaries	1358848	7.50%	1040160	4.77%
strinpool	1087944	6.01%	1087944	4.99%
declarations	1605232	8.86%	2115094	9.70%
types	411400	2.27%	318000	1.46%
dataflow:				
SSA names			1497560	6.87%
var annotations			453760	2.08%
SSA operands			1743512	7.99%
PHI nodes			336544	1.54%
control flow:				
CFG	1948344	10.76%	1854808	8.5%
cgraph	165672	0.91%	163280	0.75%
initialization:				
optabs	1361424	7.52%	1361424	6.24%
builtins	310016	1.71%	310016	1.42%
rest:				
unanalyzed	1313321	7.25%	1826243	8.37%
overall	18110289		21811905	

Figure 5: Memory usage of `combine.c`

	pre-IPA bytes	rel	post-IPA bytes	rel
statements:				
trees in statements	28126144	9.57%	26980304	5.08%
stmt annotations	14135616	4.81%	12762528	2.4%
stmt lists	8521680	2.90%	7460984	1.4%
declarations:				
labels	4433400	1.51%	16548090	3.11%
temporaries	9165376	3.12%	5254656	0.99%
strinpool	4412832	1.54%	587072	0.86%
declarations	90714368	30.87%	85981080	16.18%
declaration copies			102670637	19.32%
types	36051816	12.27%	50381072	9.48%
scope blocks	?	?	100402536	18.89%
struct function	8944640	3.04%	?	?
dataflow:				
SSA names			7805272	1.47%
var annotations			5590560	1.05%
SSA operands			19958434	3.76%
PHI nodes			1166120	0.22%
control flow:				
exception handling	4375088	1.49%	2573056	0.48%
CFG	21463944	7.30%	13346016	2.51%
cgraph	6643664	2.26%	1346296	0.25%
initialization:				
optabs	1204424	0.41%	1204424	0.23%
rest:				
C++ parser	15504496	5.28%	24133576	4.54%
unanalyzed	40176731	13.67%	65366421	12.3%
overall	293874219		531385559	

Figure 6: Memory usage of DLV testcase

	function at a time and inlining	unit at a time	cross module	whole program	whole program and profile
gzip	0.63%	1.71%	4.15%	1.35%	7.21%
vpr	0.68%	1.46%	2.05%	1.36%	4%
gcc	0.59%	3.66%	2.05%	2.56%	11.94%
mcf	0%	0.48%	0%	-0.95%	-0.32%
crafty	4.92%	6.22%	6.49%	2%	11.74%
parser	0.35%	5.6%	5.48%	6.18%	31.86%
eon	14.55%	14.55%	14.86%	14.91%	14.96%
perlbmk	1.16%	1.09%	2.76%	2.61%	15.59%
gap	0%	0%	-0.75%	-0.5%	7.81%
vortex	-1.82%	2.79%	10.02%	12.88%	34.63%
bzip2	0.37%	1.47%	1.56%	5.22%	8.42%
twolf	-0.26%	-0.17%	0.6%	2.06%	6.62%
Geomean	1.68%	3.17%	4.02%	4.04%	12.45%

Figure 7: Subset of SPECint2000 results on AMD Opteron. Performance improvements relative to function at a time compilation with no inlining.

	function at a time and inlining	unit at a time	cross module	whole program	whole program and profile
gzip	0.2%	-1.56%	-0.4%	-1.36%	-2.64%
vpr	0.47%	0.27%	1.49%	-1.03%	-1.09%
gcc	1.45%	9.93%	26.49%	19.33%	4.52%
mcf	0%	0.25%	0.52%	-0.49%	0.52%
crafty	0.73%	1.4%	5.79%	5.38%	5.78%
parser	0.04%	8.02%	7.69%	4.95%	6.57%
eon	0.39%	0.39%	0.39%	0.39%	0.39%
perlbmk	0.57%	2.86%	13.41%	10.62%	0.72%
gap	2.16%	5.16%	10.04%	8.85%	5.3%
vortex	0.21%	-1%	8.27%	-6.71%	-11.17%
bzip2	0.55%	1.12%	2.28%	-0.15%	0.42%
twolf	0.38%	1.32%	2.97%	-0.51%	0.03%
Geomean	0.59%	2.29%	6.34%	3.06%	0.67%

Figure 8: Code size growths (for x86-64) relative to function at a time compilation with no inlining