Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon Incorporated*
C. Craig Ross, *Linux Symposium*

Review Committee

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

Proceedings Formatting Team

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

Interprocedural optimization on function local SSA form in GCC

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

Abstract

GCC is slowly shifting towards interprocedural and intermodule optimization. The paper describe experimental implementation of interprocedural optimization on single static assignment form (SSA) and give an guide to writing SSA aware interprocedural optimization pass using the new framework. Some simple improvements allowed by SSA optimizations are implemented and benchmarked to discuss pros and cons of interprocedural optimization on SSA compared to current GCC non-SSA implementation.

1 Introduction

The GCC compiler has a pretty mature intraprocedural optimizer framework that, despite the high number of different architectures GCC can target, is able to compete well with proprietary solutions that are specialized for single architectures.

Instead, the GCC interprocedural optimization framework is in its early stages. Several optimizations passes were implemented with varying success, but the limitations of the framework are clear, and more work is necessary in order to make the design of passes easier (partly addressed by this paper) and to make whole framework more scalable.

In this paper we consider reorganizing the interprocedural optimizations to work on the functions in single static assignment form (SSA form from now, see [Cytron91]) in order to strengthen the analysis and improve the flexibility of optimization passes.

One of major drawbacks of GCC interprocedural framework is the memory consumption caused by the GIMPLE intermediate language. Memory issues are so serious and are considered a major showstopper towards implementing link-time whole program optimization. While in our work we are not attempting to solve the memory issues, we are trying to make it no worse. There are SSA based intermodule optimizers in existence, such as LLVM [Lattner03] with significantly lower memory footprint, proving that this approach makes sense for a production compiler.

2 Design overview

During the tree-SSA project [Novillo03], the GCC intraprocedural optimization framework was reengineered to allow multiple intermediate languages and progressive lowering, instead

of optimizing directly on a very low level intermediate language (RTL).

Front-ends now are supposed to produce GENERIC, a high level intermediate language similar to C parse trees. GENERIC is then lowered to GIMPLE, a restricted subset of GENERIC that is essentially a three-address intermediate language, represented as trees and still with source level control flow representation. Further transformation include lowering of control flow statements into conditional jumps (with a control flow graph on top), and later construction of the SSA form. Later on, GIMPLE is brought back out of SSA form and function bodies are transformed into the RTL intermediate language used by code generation.

Despite a theoretical possibility to have optimizations working on different levels of GIMPLE, all tree-level intraprocedural optimizations done now (except for constant folding and basic control flow cleanups) are performed on GIMPLE in SSA form with control flow graph built. In contrast, the interprocedural optimizations (profiling, constant propagation, cloning, inlining and alias analysis) were performed initially on GENERIC, and now on low level GIMPLE before conversion to SSA form.

This design is believed to have smaller overall memory footprint, since the function bodies are expanded one at a time into the more memory intensive SSA form. On the other hand this imposes restriction on the pass ordering, since it is impossible to run any intraprocedural optimization passes before interprocedural optimization is finished.

3 Implementation details

The SSA data structures were modified to allow multiple functions in SSA form at once. To accomplish this, most of global variables holding SSA form of function have been localized to the `struct function` structure used to hold other function specific data. Some of datastructures don't localize easily however.

3.1 SSA version numbers

The SSA versions (names) of each variable are represented by unique tree nodes that are assigned unique integers (version numbers). These numbers are used not only for debug output, but also by optimization passes to store pass specific data in on-side arrays and bitmaps. This means that the numbers must be reasonably dense.

In the current implementation the SSA version numbers are kept local to each function. This means that interprocedural passes dealing with SSA versions from different functions at a time needs to keep hashtables based on the addresses of the SSA names instead of arrays. If this become an issue in future, either the local passes would need to be moved from arrays to hashtables or two SSA version numbers (global and local) would need to be assigned to each variable.

3.2 Referenced variable numbers

Similarly to version numbers, information on the referenced variables used to be kept in an array, so that local information could be easily attached. We changed this array to a hash table in the mainline compiler, at the beginning of our project.

3.3 Variable annotations

A lot of information about variables is stored in so-called variable annotations—structures

pointed to by direct pointer from variable declarations. The annotations currently hold both pass local data and data passed across optimization passes. Unfortunately the variable annotations can not be easily privatized, since annotations of global variables needs to be shared across multiple functions.

Actually, annotations mostly hold information that is local to optimization passes (or passed from a pass to subsequent one), so that it can easily be shared by intraprocedural passes on different function bodies. In fact the only field that was made private is `default_def`.

Aliasing information is also stored in the variable annotations, and is local to functions. Since the optimizer currently is not building any aliasing information in the early optimization passes, this does not cause any issues. (Aliasing information would also consume too much memory because of the virtual operands. It is probably impractical to build it for whole compilation unit). In future however we probably want to make the early optimization passes to take aliasing into account, at least in a limited fashion, so these issues should be resolved in longer term.

The aliasing information should probably be moved to separate datastructure allocated only for pointers, where it is actually used. This should conserve memory as well as make the representation more flexible. However, the only mean to assign the datastructure to variable currently is an hashtable. This may prove too slow because aliasing info is accessed frequently during the operand scan.

From the memory consumption standpoint, the variable annotations are poorly designed, since the lifetime of temporary objects is extended across whole SSA optimization queue. Since the annotations accounts roughly of 7% of overall memory consumption, we also suggest trying to move as much as possible out of the annotations, either eliminating them completely, or keeping only the global information that need to be computed for each variable in the function. Ironically no such information is stored currently in the annotations at all and it seems also natural to store such information directly in the variable declarations.

4 Pass manager

The GCC pass manager was extended to allow interprocedural passes. The toplevel pass queue now consist of interprocedural passes, while the subpasses are considered to be intraprocedural (the passmanager automatically takes care to execute the subpasses for each function in compilation unit). In future this might be relaxed to allow interprocedural subpasses but there is no reason for doing so at the moment.

Earlier implementation of interprocedural optimization in GCC were performing analysis of all functions for all interprocedural passes first and later applying the results of interprocedural analysis on each function locally. Each pass had defined `analyze` method called for each function, `execute` method called once all functions was analyzed and finally `modify` method called again for each function separately. This was believed to allow more scalable intermodule optimization.[1]

This scheme, however, turned out to be difficult to manage because of the interactions between passes. It seems unnecessary to introduce such a restriction on interprocedural pass

[1] If all analysis are performed before any modification, the analysis phase can be done locally at compilation time and written into fake object files in function summaries. The link-time optimizer then can read the summaries first and perform interprocedural propagation of collected data. Compilation then can be done at function basis with loading only the necessary function bodies into compiler memory reducing peak memory usage. See [Hubička04] for more discussion on the topic.

designers in such an early stages of development on the framework. Once the implementation is sufficiently mature and we have better understanding of the interactions of individual passes, we might revisit this idea. It might be however more profitable to simply forget about it and concentrate more on reducing memory usage of our intermediate language.

5 Order of optimization passes

At present the interprocedural pass queue includes the following passes:

1. Removal of unreachable functions and variables.
2. Early inlining.
3. Early intraprocedural passes (profiling, control flow graph cleanup, conversion to SSA, constant propagation, value range propagation and dead code removal) and rebuilding the callgraph edges.
4. Interprocedural constant propagation and function cloning.
5. Inlining
6. Removal of unreachable functions.
7. Alias analysis.
8. Type escape analysis.
9. Points-to analysis.
10. Intraprocedural passes and final output of function.
11. Output of static variables still referenced by optimized code.
12. Local optimization and output of individual functions.
13. Output of static variables still referenced by the optimized function bodies.

The order is generally natural, perhaps with the exception of early inlining pass. Early inlining was introduced to help C++ testcases with high abstraction penalty and employs simplified heuristics, whereby we only inline functions that are smaller than the expected call overhead. In particular, wrapper functions are eliminated reducing significantly (by more than factor of 10) the profile instrumentation overhead on the TraMP3d testcase [TraMP3d]. The pass should also improve the effectiveness of early local optimization passes on testcases with a lot of small functions, where these passes are otherwise close to useless.

Interprocedural constant propagation is performed before inlining so that the inliner can deal more aggressively with the specialized bodies of functions. Unfortunately at the time of writing the paper, the interprocedural constant propagation is limited to create at most one clone of each function body (and only in the case the operand is the same constant in all calls to the function but would not be propagated without cloning because function might be called externally), which makes the interaction with the inliner suboptimal. The really interesting case of function called with different constant operands form different places specialized into multiple forms is not considered for the moment.

To enhance the effectiveness of interprocedural alias analysis, it might be also be profitable to redo early local optimization passes before it. However, the lack of local alias analysis information in early optimization passes causes any statement accessing memory to be considered volatile and thus left unoptimized. As a result, early optimizations won't improve code enough to make any difference for aliasing analysis.

6 Writing an interprocedural GCC pass

Thanks to the new pass manager framework, the interface to interprocedural passes is very similar to interface to local optimization passes. The `tree_opt_pass` structure needs to be filled in:

```
struct tree_opt_pass my_pass =
{
  "pass_name"
  gate_function,
  execute_function,
  NULL, NULL,
  /* Local subpasses */
  0, /* Static pass number. */
  TV_PASS, /* tv_id */
  PROP_cfg | PROP_ssa,
  /* properties_required */
  0, /* properties_provided */
  0, /* properties_destroyed */
  0, /* todo_flags_start */
  TODO_dump_cgraph
  | TODO_dump_func,
  /* todo_flags_finish */
  0 /* Used by RTL passes */
};
```

The function `gate_function` is used to control execution of the pass and `execute_function` is called when the pass should be performed. The pass then needs to be registered into optimization queue in `passes.c`.

6.1 Walking functions in program

To analyze functions, one should look at linked list of callgraph nodes. The list contains all functions, external or internal. To explore only the functions whose body is known, the flag `analyzed` needs to be tested:

```
struct cgraph_node *node;
for (node = cgraph_nodes; node;
     node = node->next)
  if (node->analyzed)
    ....
```

6.2 Walking callgraph

The callgraph edges (call sites) are represented as linked lists of `struct cgraph_edge` objects. Each callgraph node points to list of callers by `node->callers` and list of callees by `node->callees`. These lists are built during analysis pass and are maintained up to date until final local optimization passes that are destructive to callgraph.

6.3 Walking function bodies

Most functions that manipulate control flow graph or SSA form are not yet aware of multiple functions. When the control flow graph needs to be manipulated, it is easiest to change the `current_function_decl` and `cfun` pointers. It is also necessary to setup the control flow graph hooks:

```
push_cfun
 (DECL_STRUCT_FUNCTION
    (node->decl));
tree_register_cfg_hooks ();
current_function_decl
 = node->decl;
```

After the analysis is completed, it may be necessary to destroy dominance info (if computed) since this datastructure is global.

```
free_dominance_info
 (CDI_DOMINATORS);
free_dominance_info
 (CDI_POST_DOMINATORS);
pop_cfun ();
```

If the transformations affected the callgraph, it is necessary to update the callgraph datastructure, either by hand or via `rebuild_cgraph_edges`. If substantial changes were made to the function body, it might be profitable to re-do early optimizations by `tree_early_optimization_passes` that subsume `rebuild_cgraph_edges`.

6.4 Walking variables

Variables are, similarly to functions, grouped in a linked list:

```
struct cgraph_varpool_node *vnode;
for (vnode
       = cgraph_varpool_nodes_queue;
     vnode;
     vnode = vnode->next_needed)
  analyze_variable (vnode);
```

7 Experimental results

In this section we compare the memory consumption, compilation time and quality of produced code of our experimental branch compared to mainline compiler from date of last merge to the branch (2006-04-06).

7.1 Cost of SSA form

The SSA form and is more memory intensive because of the `SSA_NAME` wrappers and other datastructures. We made no effort to reduce memory usage of the SSA form and merely localized all datastructures needed. Still, the negative effect is slightly outweighed by the scalar cleanups (constant propagation, value range propagation, copy propagation, forward propagation and dead code removal) performed just after converting to SSA form.

On TraMP3d testcase (consisting of large number of tiny function), the memory consumption increase after the lowering passes is 211 MB compared to 209 MB on mainline compiler. This testcase should expose near to worst-case behavior since it consists of number of very tiny functions. On `combine.c` (GCC module) testcase, the memory is 8.4 MB compared to 8.5 MB for mainline compiler. It looks like the simple cleanups enabled by SSA conversions are effective enough to pay back the extra memory cost.

7.2 Cost of performing inlining on the whole compilation unit

The extra memory cost of moving the inline transformation early as discussed in Section 4 is measured by introducing garbage collector pass just after the inliner pass. This way, we can compute how much memory is still referenced.

On the IPA branch, the memory consumption for TraMP3d testcase just after inlining peaks at 390 MB, while on mainline it is only 184 MB. Again, TraMP3d should expose near to worst case behavior. There are no noticeable differences for `combine.c` testcase since very little inlining is performed there.

While this memory increase seems serious, it can be argued that it is bound by overall unit growth parameter of the inliner and thus should not lead to uncontrolled memory consumption problems. However, we felt it was too early to change inlining order in GCC 4.2, and decided to delay its submission for later version when we can use the extra flexibility introduced by the change.

It is trivial to modify our experimental branch back to mainline behavior in this case and it can be easily verified that the overall memory peaks of both compilers are same then.

7.3 Overall compilation time

Compilation time is affected by multiple factors. Obviously the optimization queue was extended by another 4 scalar optimization passes, that should account together by about 0.5–2% of compilation time. Additionally, the inliner now has to maintain the SSA form, with some cost in compilation time.

However, the early optimizations save work after inlining and can reduce the memory footprint of the function body just after the first alias analysis pass. Thus, overall compiler performance is sped up for some testcase, as can be seen by decrease in overall allocation of GGC memory in both `combine.c`. As a result, the compile time performance tests show a pretty mixed picture. `combine.c` compiles about 2% slower, but the whole bootstrap is a few percent faster, because early optimization is effective on `insn-attrtab.c` and saves a lot of memory for the aliasing information.

For the TraMP3d testcase, compilation time benchmarks are not directly comparable because the inliner decisions are significantly different. The experimental compiler seems to be about 4% slower because of more aggressive inlining; on the other hand, the resulting binary is both smaller and slightly faster.

The overall compilation time of SPECint benchmarks remains about the same for single file optimization and improve from 510s to 490s for whole program optimization.

8 Runtime performance

The SPECint 2000 benchmark results (except for twolf benchmark being misscompiled) of mainline compiler are compared to the experimental branch in Table 1. The tests was compiled with `-O3 -ffast-math` optimization setting. Three levels of interprocedural optimizations are considered: with single file optimization, pseudo-link-time optimization (where all the source files are parsed first, via `--combine`, and optimized as single compilation unit) without and with whole program assumptions (via `-fwhole-program` that make all functions and variables local with the exception of `main()`. GCC is not able to combine C++ modules into single compilation unit and thus eon benchmark (only C++ benchmark in SPECint) is always compiled with the same settings.

For broader picture, the comparison to Intel C++ compiler (ICC 9.0) is included too. The single file optimization benchmark was performed with `-axW -ip -O3`, while whole program with `-axW -ipo -O3`. It shall be noted that ICC is not able to tune for AMD chips and thus suboptimal pentium4 tuning was used instead. ICC is able to do full linktime optimization for eon benchmark too.

The results are not surprising. The branch brings up to 2% improvements, almost entirely because of improved inliner decisions in the crafty benchmark, and improved aliasing in the gcc benchmark. Both are caused by early optimization. The improvements are mostly visible in link time optimization, without whole program assumptions that (even on mainline) simplify the inlining of functions that are called just once in whole program.

Noticeable improvements are also in overall code size savings of roughly 117 Kb, 467 Kb, 382 Kb[2] from overall size of binaries (compiled with single file, link time and whole program setting accordingly), suggesting that inliner is able to make better decisions in both performance and code size metrics.

[2]The SPEC binaries compiled with dynamic linking and without debug info are about 5MB.

Author also measured 7% runtime speedup of TraMP3d benchmark but due to its overall instability relative to inliner decisions, this might be just "luck" (earlier version of compiler had opposite results). It can be expected however, that for more complex (and/or less hand optimized) testcases than SPECint benchmarks, the benefits will be more noticeable.

9 Conclusion

As expected, the new SSA based intermodule framework improve the runtime performance of some benchmarks that are sensitive to inlining. The benefits come mostly from pre-inline optimization passes and are supposed to increase as the interprocedural optimization is made more effective.

Surprisingly, the memory consumption problems caused by transition to SSA seems to be no worse than with the current (non-SSA) framework. Performing inlining before other interprocedural passes, however, causes significant memory consumption growth—which is fortunately limited by overall unit growth limit of inliner. So far, this change is however independent on the rest of work on IPA branch, so both issues can be considered separately.

Sadly, memory consumption in mainline GCC is considered to be one of the major showstoppers on the way towards intermodule optimization and radical reductions will be necessary in future. Perhaps, completely switching to a new intermediate language is necessary, as discussed in [Lattner03] or developing different memory representation of GIMPLE not based on nested tree structure.

10 Acknowledgments

The plan for moving interprocedural optimization into SSA form and was discussed at the 2005 GCC Summit with Daniel Berlin, Diego Novillo, and Kenneth Zadeck. Daniel Berlin contributed patch to avoid `referenced_vars` array. Razya Ladelsky adapted interprocedural constant propagation to work on SSA form. Olga Golovanevsky contributed a devirtualization pass. Paolo Bonzini proofread the paper, helped to clarify it, and significantly reduced the amount of Czenglish.

References

[Cytron91] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck, Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Transactions on Programming Languages and Systems 13,4* (1991), 451–490.

[Novillo03] D. Novillo, Tree SSA — A New Optimization Infrastructure for GCC *Proceedings of the 2004 GCC Developers Summit* (2003), `http://www.gccsummit.org/2003`

[Hubička04] J. Hubička, Call graph module in GCC, framework for inter-procedural optimization, *Proceedings of the 2004 GCC Developers Summit* (2004), `http://www.gccsummit.org/2004`

[Lattner03] C.A. Lattner, Architecture for a Next Generation GCC, *Proceedings of the 2004 GCC Developers Summit* (2003), `http://www.gccsummit.org/2003`, `http://www.llvm.org`

[TraMP3d] Template heavy C++ testcase
available at:
http://www.suse.de/
~gcctest/c++bench/

Benchmark	single file			linktime		whole program		
	non-SSA	SSA	ICC 9.0	non-SSA	SSA	non-SSA	SSA	ICC 9.0
gzip	1206	1214	1225	1188	1210	1158	1175	1323
vpr	858	855	857	848	848	864	864	859
gcc	1072	1050	960	1074	1077	1101	1100	980
mcf	539	543	543	543	541	543	544	543
crafty	1944	2066	2100	1864	2116	2086	2097	2211
parser	828	829	801	826	832	836	830	811
eon	2414	2473	1803	2422	2477	2422	2478	2033
perlbmk	1479	1460	1419	1407	1472	1464	1470	1513
gap	1110	1147	1085	1174	1162	1194	1210	1094
vortex	1708	1737	1626	1824	1859	1857	1905	1917
bzip2	1020	1020	1005	1021	1019	1058	1072	1011
Geomavg	1188	1200	1138	1189	1214	1214	1227	1194

Table 1: Runtime performace (in SPECint ratios, bigger is better)