Reprinted from the

Proceedings of the
GCC Developers' Summit

June 2nd–4th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
Janis Johnson, *IBM*
Toshi Morita, *Renesas Technologies*
Zack Weinberg, *CodeSourcery*
Al Stone, *Hewlett-Packard*
Richard Henderson, *Red Hat, Inc.*
Andrew Hutton, *Steamballoon, Inc.*
Gerald Pfeifer, *SuSE, GmbH*

Proceedings Formatting Team

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

The GCC call graph module

a framework for inter-procedural optimization

Jan Hubička
SUSE ČR
jh@suse.cz

Abstract

The implementation of call graph based optimizations in GCC required several design changes to the interfaces in between front-ends and back-end parts of the compiler. We describe in detail the new interfaces, optimizations we implemented (in-lining and basic inter-procedural propagation) and the call graph datastructure itself. We compare memory consumption, compilation time and code quality of function at a time and unit at a time compilation scheme. We also outline future plans for the more advanced inter-procedural optimizations and whole program optimization.

1 Introduction and motivation

The implementation of function inlining in gcc used to be a major source of dissatisfaction among users of the compiler. Even though inlining had been redesigned from scratch in GCC 3.0, both inliners had serious problems.

The old inlining implementation (based on the low-level RTL intermediate language) could not remove several ugly artefacts in the code, such as in-memory structures used to pass arguments. It also consumed unnecessarily large amount of memory to store function bodies in RTL form. Memory consumption was further increased by storing functions after inlining of callees instead of before.

The new tree-based implementation of inlining in GCC 3.x solved all of these problems but unfortunately brought several new issues. For very complex C++ programs, the new inlining decision heuristics inlined too many functions causing extreme memory consumption, large compile times, and impractically bloated applications. On the other hand the default inline limits were way too low for C programs such as the Linux kernel, causing many functions to not be inlined at all despite the programmer having manually marked them inline. As a result compiler became almost unusable for some C++ programmers working on template heavy code (such as POOMA library) and Linux kernel developers adopted the paradigm of using the `always_inline` attribute to override the default inlining heuristics everywhere.

In addition to these problems, GCC traditionally was unable to perform "backward inlining" (inline functions used before defined), causing noticeable loss in some benchmarks such as SPEC2000 when compared to other compilers.

It seemed impossible to tune the inlining heuristics using the available set of parameters, and thus we started to look for a more involved solution. While looking at the problem from a high level, it seems to be really easy to simply inline all "small" functions as long as doing so does not cause "extreme bloat." Defining which functions are "small" can be done easily by limiting number of instructions in it's body, while defining code bloat can be done with two parameters: first one limits growth of single function body (since compiler algorithms are generally not linear, and for really large functions, produce both poor code and long compilation times) and the second one limits growth of overall binary size. Unfortunately without whole program optimization (still mostly out of reach of the current GCC framework), it is impossible to realize the last argument, but one still can limit the overall growth of single compilation unit and get similar results.

Because implementation of such a global parameters for function inlining was very difficult with the original organization of the compiler we took a more difficult path and first developed an infrastructure to assist inter-procedural optimization, to be used later when focusing on the inlining issues.

In this paper we describe the infrastructure and the new optimizations implemented while working on this project. The rest of this paper is organized as follows. In Section 2 we briefly describe some problems we had to deal with and solutions we chose for them; in Section 3 we describe the basic data structures we use; in Section 4 we describe the interface to the front-end; in Section 5, the implementation of inlining; and Section 6 contains some experimental evaluation of the new algorithms.

2 Overall design and the implementation challenges

GCC compiled the majority of functions immediately after parsing their bodies (only a few functions, such as static inline functions, were special-cased and deferred until it was obvious that the out of line copy is needed) making implementation of inter-procedural optimizations impossible. It was necessary to reorganize the compilation process in a way so all functions are parsed first, then analyzed and compiled last. We will refer to this scheme of compilation as *unit-at-a-time* as opposed to function-at-a-time used by GCC originally.

The main problem that arised was that the original GCC design made it very difficult to change the compilation order. The back-end has been organized as a library that allowed the front-end to compile a specified function. Each of the front-ends implemented its own (in some cases remarkably complex) logic on compiling and/or deferring a function and expected the compilation to happen immediately after passing it to back-end (for instance, the C++ front-end looked back into the symbols actually output to the assembly file to figure out which functions were referenced and had be compiled).

Instead of implementing unit-at-a-time logic into each individual front-end, it seemed easier to reorganize the interface in between the front-ends and back-ends to allow implementation of the generic compilation driver taking care of all the decisions. Since reorganizing all the front-ends at once was a difficult task, the new API has been made optional, and we first implemented unit-at-a-time for the C front-end only and later started work on reorganizing the others.

At the moment, only the C, Objective C, C++, Java, and F90 front-ends have been updated to the new API, and with exception of C, each conversion was a nontrivial task. C++ needed to look back into assembly files to discover what templates needs to be instantiated; Objective C gathered information about method API during compiling the function body, and later producing functions using that information; and F90 and Java use trees slightly different from the C++ family, and broke some expectations in the new code.

Switching to unit-at-a-time by default just seemed too radical. The main concerns that were pointed out in the discussion about the change were about peak memory usage growth: in function-at-a-time mode, the function bodies can be released early once the processing of given function finished, while unit-at-a-time mode needs to store into memory all functions at once. If the amount of memory occupied by the function bodies gets too large, it may result in slow down of the compilation.

As a result of this discussion, we decided to allow coexistence of both schemes and added the command line option `-funit-at-a-time` to choose particular one. To date, optimization levels `-O0` and `-O1` by default use function-at-a-time compilation, while `-O2` and `-O3` use unit-at-a-time. Once the front-end is converted into the new API, both supported compilation schemes (unit-at-a-time and function-at-a-time) appear almost identical to the front-end, and all the logic is hidden in the new compilation driver implemented in `cgraphunit.c`.

The compilation process is now organized as follows:

1. Parsing phase: This step is fully controlled by the front-end. It is up to the front-end to decide when a given function is "finalized" and pass it to the compilation driver. After that point the front-end is not allowed to make any modifications on the function body or declaration, and it is fully up to the compilation driver to decide when (and if) the function will be compiled.

   It is probably important to note that there is one exception the rule disallowing any changes to the functions passed to the back-end. The C front-end, GCC allows the function to be first defined as `extern inline` and later be re-defined with a completely different body as an ordinary function. In this special case, we allow the finalization to be called twice; we simply remove all traces of the old body from the data structures and mark the function as uninlinable, then, when this situation is detected.

   At this stage, early analysis of finalized functions is done as well. Certain warnings (such as about unused function parameters) are output here, since it is the last time we'll see unneeded functions. It is also decided whether the function is an "entry point"—i.e., whether it is reachable from unknown code by some way (such as via external linkage).

   The difference between function-at-a-time and unit-at-a-time mode also lies in the finalization code. In unit-at-a-time mode, the function is just stored into the data-structure and left for later analysis, while in function-at-a-time mode all functions are fully analyzed immediately, the control flow graph is incrementally built, and most functions are compiled—the only exceptions being static inline, extern inline, comdat,[1] and nested functions. These are just stored into the call-graph and compiled only when they turn out to be necessary (i.e., when symbol is output into the assembly file).

   A similar mechanism is implemented for file-scope variables. In unit-at-a-time, all variables are stored into variable pool data-structure, while in function-at-a-time mode, all variables are output to the assembly file immediately.

   In function-at-a-time mode compilation

[1] functions that may appear in multiple units and are linked into a single function.

terminates once parsing is finished, while in unit-at-a-time it goes into following stages:

2. Analysis phase: The call-graph is built and local optimization information is gathered at this stage. To reduce the amount of work done, the call-graph is built incrementally and only functions reachable from the entry points are analyzed. Since we do not handle any dependency edges on data-structures, the reachable data-structures are immediately output into the assembly file and further functions/data structures referenced by them are added into the work lists via a callback from back-end function, outputting a symbol reference into the assembly file.

   The local analysis used to drive interprocedural optimizations is also supposed to happen here. At the moment, the size of function body is estimated for later use in inlining.

3. Optimization phase: Several optimizations are performed on the call-graph itself in sequence. At the moment following optimizations are done:

   (a) Reclaiming of memory occupied by the unused (i.e., unanalyzed) functions and data-structures.

   (b) Local function discovery: A *local function* is a function that is not an entry point and whose address has never been taken. We mark these functions by special flag, since it is possible to perform optimizations interfering with the target ABI on such functions. For instance on i386 we now use register-passing conventions, but there are considerably more possibilities for target-specific optimization here. (In PIC compilation, one can, for instance, avoid recomputing of global offset table pointers in the prologues of local functions, and propagate the computation into callers.)

   (c) Construction of inlining plan: We make all the inlining decisions in advance and store them in call graph as a so-called "inlining plan." See Section 5 for details.

   (d) Another pass of unreachable function removal: in some cases, a function might be reachable only via a call in an extern inline function that was never inlined. Since the body of the extern inline function is never output, it is possible to remove all such functions, too. This scenario is very common for C++ programs.

   Note that it is very desirable not to touch the function bodies at this stage. In real whole program optimization, the functions are parsed and stored into "object files" containing intermediate representation of the program. The intra-procedural optimization phase executed in linker then should not need to load everything into memory at once and instead use the data files as a database reading the call-graph information first and using the function bodies just later in the compilation phase.

4. Expansion: We proceed in reverse DFS order on functions that are still present in the call-graph, applying inter-procedural optimizations such as inlining to the functions, and finally leaving them to the back-end to do the actual optimization and compilation.

   Function reordering allows more reliable propagation of information from the callee code generation into the caller. For instance, it is possible to generate a better call sequence when the callee's preferred stack frame boundary is known.

Such function ordering would permit implementation of more interesting optimizations too (for instance simple interprocedural register allocation). On the other hand, it makes it almost impossible to avoid compilation of some function when its call has been optimized out. At the moment we make no attempts to solve this issue; however, in the future we may want to do early optimization during the analysis stage to catch most of these cases.

It also would be also desirable to defer output of global variables to this stage and output only the variables that are still referred by functions after the optimization. Implementing this feature is easy and we hope to do so in the near future.

3 Data-structures

Most of the code in the compilation driver actually manipulates only two data structures, that is, the call-graph and the variable pool.

3.1 The call-graph

The *call-graph* consist of nodes and edges represented via linked lists. Each function (external or not) corresponds to the unique node and each direct call has corresponding edge from caller to the callee.

The mapping from declarations to call-graph nodes is done using an hash table based on the declarations' `DECL_UID`, so it is essential that the frontend use single declaration ID for each function or variable. The call-graph nodes are created lazily using the `cgraph_node` function, when an unknown declaration is called.

When the call-graph is built, there is one edge for each direct call. The indirect calls are not represented at a moment. We simply mark each function with address taken as externally visible function. Optimizers then have to expect conservatively that each indirect call and/or call of unknown function might in turn call some of the entry points. The entry points are merged via flag `needed` in the call-graph node.

Finally there is a work list used to maintain nodes that are reachable from the entry points and thus needs to be analyzed or output into the file.

3.2 Data-structures for inter-procedural information

Call-graph is place to store data needed for inter-procedural optimization. All data-structures are divided into three components: `local_info` that is produced while analyzing the function, `global_info` that is result of global walking of the call-graph on the end of compilation and `rtl_info` used by RTL back-end to propagate data from already compiled functions to their callers.

The division has been made to make it possible to reduce memory usage in the future. Each of the field has different lifetimes and thus they don't necessarily need to be allocated all the time. At the moment the data-structures are small and thus all allocated at once with the call graph nodes, but the `cgraph_global_info`, `cgraph_local_info`, `cgraph_rtl_info` accessor functions shall be used to access the data. These functions already contain sanity checks that enforce the lifetimes of the individual data structures.

In the contrast, there is structure `function` allocated for each parsed function body traditionally used to store related information by many other parts of the compiler. This structure has no such organization and it consumes up to 25% of overall memory for some C++ programs. We hope to improve the situation by reorganizing `struct function` similar way and moving to the call-graph nodes some of the data currently held in `struct function`, removing redundancies on where the information shall be stored.

3.3 The varpool data structure

In order to allow elimination of unused static data within the backend, we modified the interface to the output data-structures too. The varpool module is used to maintain variables in similar manner as call-graph is used for functions. At the moment it is implemented as a simple hash table containing entries for all global data-structures, and a worklist maintaining a list of variables that need to be output into assembly file. No dependencies or references are represented explicitly.

4 Front-end API

An important part of the new compilation driver design is the API to front-end. We tried hard to make it as easy to use as possible, however practice has shown that it is not always trivial to update existing front-ends to the new philosophy. Hopefully the API will still be natural to use in the new code.

All functions the front-end programmer shall be interested in are:

cgraph_finalize_function shall be called once front-end has parsed whole body of function and it is certain that the function body nor the declaration will change.

(As mentioned above, there is one exception needed for implementing GCC's `extern inline` functions, but it should not be used by new code.)

cgraph_varpool_finalize_variable has the same behavior but is used for file scope variables.

cgraph_finalize_compilation_unit shall be called called once parsing of compilation unit is finalized and trees representing it will no longer be changed by the front-end.

In unit-at-a-time mode, call-graph construction and local function analysis takes place here. Bodies of unreachable functions are released to conserve memory usage.

The compilation unit in this point of view should be compilation unit as defined by the language—for instance the C front-end allows multiple compilation units to be parsed at once and it should call this function each time parsing is done, in order to save memory. This is not what happens currently because the C front-end does global static variable renaming pass at the very end of compilation. As a result, unnecessary and duplicate function bodies are maintained in memory up to very end of the parsing process.

Modifying the C front-end to use this scheme is not an easy task. Merging of C compilation units together involve a lot of C language specific behavior and we need to consider whether it is feasible to implement that logic in the generic pass or through a some simple set of front-end hooks.

cgraph_optimize performs inter-procedural analysis and compile functions in unit-at-a-time mode (in function-at-a-time this function does nothing except for producing debug dumps). Front-end shall call this function at the very end of compilation, after releasing all those internal data-structures that are not passed to the back-end.

cgraph_mark_needed_node can be used when a function is referenced by some hidden way (for instance if it is marked by attribute `used`, which usually means that it is called in inline assembly code). The call-graph data structure is updated in a way that function is marked as entry point and thus it is never optimized as local function and always compiled.

cgraph_varpool_mark_needed_node has a similar meaning as function `cgraph_mark_needed_node`, but is used for variables.

To overcome problems in the front-end specific representation of trees, we had to implement two callbacks that allow a front-end to define front-end specific expansion of trees into RTL. We plan to eliminate these completely once the work on tree-ssa branch is finished.

analyze_expr callback This function should lower tree nodes not understood by generic code into understandable ones or, alternatively, should mark referenced call-graph and varpool nodes.

expand_function callback is used to expand the function into RTL form in front-end specific way. The front-end should not make any assumptions about when this function can be called. Existence of this hook is also used as a check on whether front-end supports unit-at-a-time API.

5 Inlining Heuristics

Only non-trivial inter-procedural optimization implemented at a moment is inlining we describe in this section. The inliner implementation can be used as an example how other inter-procedural optimizers can be implemented on the on the top of the new infrastructure, so we will describe it in greater detail.

5.0.1 Inlining plans

The function inlining information is decided in advance (in the optimization phase) and maintained in the call-graph in the so called inlining plan until the function is optimized. Once a function body is physically inlined into another, the callgraph data-structure is updated to reflect new program structure. This organization is critical to make it possible to save parsed function bodies into disk and make all inter-procedural optimizations without actually touhing the bodies and having them to resist in memory all at once.

The inlining decisions are reflected in the call-graph as follows: When the heuristics decide to inline given call-graph edge, the calle's node is cloned to represent the new function copy that will be later produced by inliner (so each inlined call of given function gets unique clone node and all the clones are linked together via linked list). Each edge has an "`inline_failed`" field. When the field is set to NULL, the call will be inlined. When it is non-NULL it contains an reason why inlining wasn't performed, that might be eventually output by the inliner when `-Winline` is specified.

We originally didn't clone the nodes and simply had a flag in each edge specifying whether the given call shall be inlined. This was found soon to have many limitations. For example, it is impossible to represent inline plans that are not *transitive* (i.e., once call of function B in offline copy of function A is inlined, each inline copy of function A must have the function B inlined as well). Non-transitive inlining plans are needed in order to let the programmer claim that all direct and indirect callees shall be inlined recursively; experience has shown that this kind of control is useful in template-heavy C++ numeric code.

Reorganizing the code to new scheme also turned out to simplify significantly the estimates of overall code size growth caused by inlining, and allowed to release function body as soon as all of its inline copies are produced.

5.0.2 Profitability estimates

To make good inlining decisions, the profitability of inlining a given call must be estimated. Ideally, one might take into account the expected time spent in callee and compute how large relative speedup will elimination of the call overhead is. It is also desirable to take into account the new optimization possibilities and weight it with the expected code size growth. See for instance [1] for more discussion on the topic.

With current very high level and partly front-end specific intermediate representation it is difficult to do such a complex analysis and the profitability analysis actually represent the weakest spot of our implementation. At a moment we simply compute estimated function body size in front-end specific way by walking the tree representation and summing cost of the nodes. The majority of nodes has a cost of 1 with exception of a few nodes that are known to have zero cost (such as lexical scope regions or `__builtin_constant_p` calls) and a few others that are known to be expensive (such as division or function call) and are assigned a cost of 10. This implementation is still a noticeable improvement compared to previous implementations that were merely counting number of statements in the source and completely ignored the different complexities of individual constructs.

The cost of inlining given call is estimated as cost of increasing the callers body cost by callees cost minus 10 (eliminating the call). Our objective is to inline as many function calls before reaching given growth limits.

Toggether with developers from Apple we are working towards a better implementation of this analysis based on tree-ssa representation. This work is being done tree-profiling branch and will take into account the runtime call frequencies computed from the profile, allowing the compiler to perform a realistic estimate the costs of individual calls. We also plan to implement a partial specialization pass on functions that will notice situations where function body can be significantly simplified when some of its arguments are known. This project is however still far from being finished.

5.0.3 Limiting parameters

As discussed earlier, we provide set of parameters to avoid too extreme amount of inlining. The final set of parameters are just slightly more complicated than ones outlined in the introduction section:

max-inline-insns-single sets the maximum number of instructions (counted in GCC's internal representation) in a single function that the tree inliner will consider for inlining. This only affects functions declared inline and methods implemented in a class declaration (C++). The default value is 500.

max-inline-insns-auto sets limit on estimated size of inline candidates when `-finline-functions` (included in `-O3`) is used. The default value is 120.

large-function-insns is a limit that specifies which functions are considered to be "large": for functions greater than this limit, inlining is constrained by `--param large-function-growth`. This parameter is useful primarily to avoid extreme compilation time caused by non-linear algorithms used by the back-end. The default value is 3000.

large-function-growth specifies maximal growth of large function caused by inlining in percent. The default value is 200.

inline-unit-growth specifies maximal overall growth of the compilation unit caused by inlining. This parameter is ignored when `-funit-at-a-time` is not used. The default value is 150.

5.0.4 Global inlining heuristics

Given the rules established by these five parameters, inlining decisions are made in three passes. In the first pass all function calls marked with the `always_inline` attribute are inlined, so that other decisions cannot interfere with it.

In the second pass inlining of small functions is performed; all function candidates are put into a priority heap ordered by the estimated costs of inlining the function into all its callers and then they are inlined in priority order, updating the costs of other enqueued candidates until the heap is empty or the overall unit growth parameters reached.

This algorithm (often described as knapsack style, see [2]) seem to perform better than simple top-down and bottom-up heuristics resulting in more function calls to be inlined without breaking the same inline limits discussed above.

In the third pass all functions that are still called just once are inlined unless the callee body become too large.

Finally the fourth pass does so-called "recursive inlining." When the function contains recursive calls and its body is called, the calls are inlined up to recursion depth computed in a way so function reach size specified by parameter. This optimization has similar effect as loop unrolling.

5.0.5 Incremental inlining heuristics

The global inlining heuristics can not be used in function-at-a-time mode and thus there is an alternative implementation of simple bottom up inlining heuristics. Most of the code (checking of limits and updating call-graph) is shared in between the implementations and thus the implementation is pretty straight forward.

The major problem of this heuristics appears to be in fact that the overall compilation unit growth argument is ignored. In some extreme C++ test cases (such as those based on POOMA library) the compiler now compiles faster at `-O2` compilation level compared to `-O1`.

6 Experimental Results

Evaluating the effectiveness of new infrastructure is difficult task. The benefits (and losses) vary greatly together with the coding style of the tested application. Very good results can be measured in the template heavy C++ code, such as the DLV application or POOMA library that we use as a benchmark suite. The table 1 summarizes the results of DLV benchmark suite evolving over various GCC releases and it is easy to notice the degradation in performance in GCC 3.0, as well as a reduction of code size caused by decreasing inline limits to avoid compile time problems as mentioned earlier. This problem remained apparent until GCC 3.3 despite quite serious attempts to tune the heuristics. GCC 3.4 behaves quite well in both function-at-a-time and unit-at-a-time heuristics, but the code size has increased noticeably. For this particular benchmark it is possible to reduce the inlining limits somewhat and get code sizes smaller than GCC 2.95 without considerable performance regressions; reducing the limits, however, hurts performance in other benchmarks signalizing that the profitability analysis needs more work. Unfortunately it is no longer possible to present GCC 3.4 numbers with the old heuristics, but the initial tests did already show benefits similar as ones compared to GCC 3.3 so we believe that majority of the improvements actually come from inlining in this particular case.

The author evaluated number of template heavy test cases while working on new implementation, and the benefits can be virtually infinite scaling with complexity of the code. For test case based on POOMA library, compilation times went down from 25 minutes to 1 minute with noticeable improvements in execution time too.

On the other hand, the C and Fortran benchmarks shows a much more moderate improvement. Table 3 shows benchmarks made on AMD Opteron chip in 32bit and 64bit mode. While majority of the tests improve, the benefits are less noticeable. The good news, however, are that the unit-at-a-time reduce code size almost consistently on the `-O2` level of optimization. On the other hand the `-O3` scores demonstrate that backward inlining can cause code size growth without major changes in the performance.

By comparing the 64-bit and 32-bit scores, one also can notice the benefits of register passing conventions.

One area where author was hoping for considerable improvement is performance of desktop applications. It is difficult to present the benchmarks of the GUI application but the simple test of compiling x86-64 KDE and Mozilla source gave savings of 7.4% and 6.6% respectively in the overall size of stripped binaries, and a partial i386 Open-Office build gave 22% savings. These savings ought to bring a noticeable improvement in execution time and reduction of memory usage too. In addition the performance of code shall be improved similar way as in the DLV application benchmark presented here.

It remains to discuss the memory usage of the compiler. Again it is not difficult to present extreme improvements (for example, compiling the POOMA library only requires 2% of the memory) as well as extreme regressions: a huge compilation unit consisting of small but uninlinable functions will result in arbitrarily high unit-at-a-time peak memory usage, without increasing peak usage in function-at-a-time mode.

Real world application however show that compilation units usually require less memory, both because they are not very large and also because the lifetime data structures used by the front-end in unit-at-a-time mode does not overlap with the lifetime of data structures used by backend; in addition, unneeded functions and data-structures are released early.

Table 2 shows peak GGC memory usage while compiling some of relatively large source files. The numbers were obtained by compiling with `--param ggc-min-expand=0 --param ggc-min-heapsize=2048 -Q` and examining the GGC debug output for largest memory usage after the collection. The `generate.ii` is a large test case of template heavy code, while `combine.c` is one of largest source files of GCC. The graph of memory usage in unit-at-a-time of the C++ testcase is almost flat demonstrating that the pass releasing unneeded function bodies release enough memory so the back-end no longer increase the peak. For the C test case there are small regression at `-O1` and `-O2` but author would hope that these won't prevent unit-at-a-time from being enabled by default in the future.

7 Contributors

The project would be impossible without following contributions: Steven Bosscher reorganized f90 front-end, reviewed early implementations of the inlining code and made a number of cleanups. Richard Günther provided a lot of feedback about POOMA library issues. He also implemented patch for "`leafify`" function attribute that brought major motivation for reorganization of the inlining plans representation. Richard Henderson reviewed most of the call-graph code. Gerald Pfeifer provided the DLV benchmark that has turned out to be extremely useful to tune the heuristics and gave a lot of useful feedback. Jeff Sturm revamped the Java front-end to cgraph code. Mark Mitchell helped to choose feasible way on how to reorganize C++ compiler, reviewed the changes and helped to solve some of issues. Zack Weinberg reorganized the code to not use hash tables based on assembler names.

A number of SUSE developers (mainly Andreas Jaeger, Andi Kleen and Michael Matz) helped to test GCC on SUSE distribution build and analyzed/fixed many of compatibility issues and implementation defects so the implementation was ready for production use before the offical GCC 3.4 release.

Author would also like to thank to Paolo Bonzini and John W. Lockhart who helped to proofread the paper.

References

[1] *Towards Better Inlining Decisions Using Inlining Trials* (1994), Jeffrey Dean, Craig Chambers

[2] *A Comparative Study of Static and Profile-Based Heuristics for Inlining* (2000), Matthew Arnold, Stephen Fink, Vivek Sarkar, Peter F. Sweeney

Table 1: Speedup in the DLV Benchmark relative GCC 2.95
Execution times in second and relative comparisons to GCC 2.95, smaller is better.

benchmark	GCC 2.95	3.0.4		3.3.2		3.4	-fno-unit…	3.4	-funit-at…
STRATCOMP1-ALL	2.45s	24.92s	1017.00%	4.68s	191.00%	8.31s	339.00%	2.58s	105.00%
STRATCOMP-770.2-Q	0.49s	0.57s	116.00%	1.22s	248.00%	0.47s	95.00%	0.45s	91.00%
2QBF1	10.92s	13.96s	127.00%	28.68s	262.00%	11.06s	101.00%	9.33s	85.00%
PRIMEIMPL2	7.52s	8.75s	116.00%	43.60s	579.00%	6.27s	83.00%	6.00s	79.00%
3COL-SIMPLEX1	4.68s	4.97s	106.00%	11.13s	237.00%	4.56s	97.00%	4.34s	92.00%
3COL-RANDOM1	6.66s	8.15s	122.00%	38.14s	572.00%	5.95s	89.00%	5.86s	87.00%
HP-RANDOM1	4.93s	5.72s	116.00%	18.44s	374.00%	5.23s	106.00%	4.44s	90.00%
HAMCYCLE-FREE	0.80s	1.12s	140.00%	4.96s	620.00%	1.03s	128.00%	0.72s	90.00%
DECOMP2	8.44s	9.59s	113.00%	33.91s	401.00%	8.53s	101.00%	7.87s	93.00%
BW-P5-nopush	4.45s	4.85s	108.00%	12.90s	289.00%	4.25s	95.00%	4.19s	94.00%
BW-P5-pushbin	3.79s	4.05s	106.00%	12.61s	332.00%	3.44s	90.00%	3.40s	89.00%
BW-P5-nopushbin	1.21s	1.31s	108.00%	4.07s	336.00%	1.13s	93.00%	1.09s	90.00%
HANOI-Towers	2.05s	2.19s	106.00%	6.21s	302.00%	1.94s	94.00%	1.82s	88.00%
RAMSEY	5.34s	5.69s	106.00%	16.69s	312.00%	4.83s	90.00%	4.58s	85.00%
CRISTAL	5.30s	5.91s	111.00%	12.67s	239.00%	5.14s	96.00%	4.75s	89.00%
21-QUEENS	6.35s	7.31s	115.00%	40.15s	632.00%	5.09s	80.00%	4.86s	76.00%
MSTDir[V=13,A=40]	12.58s	14.46s	114.00%	41.77s	332.00%	9.14s	72.00%	8.60s	68.00%
MSTDir[V=15,A=40]	12.62s	14.49s	114.00%	41.44s	328.00%	9.15s	72.00%	8.53s	67.00%
STUndir[V=13,A=40]	6.47s	7.57s	117.00%	25.48s	393.00%	4.96s	76.00%	4.61s	71.00%
TIMETABLING	7.08s	7.37s	104.00%	18.21s	257.00%	6.30s	88.00%	5.90s	83.00%
compilation time	2m42s	2m53s	106.7%	2m47s	103%	2m9s	79.6%	2m28s	91.3%
Code size	1251k	622k	49.7%	1562k	124.8%	1808k	144.5%	1628k	130.1%

test	optimization level	function-at-a-time	unit-at-a-time	savings
generate.ii	-O0	33563K	32606K	2.9%
generate.ii	-O1	33462K	32606K	2.9%
generate.ii	-O2	43296K	33239K	30%
generate.ii	-O3	>55077K	33411K	>64%
combine.c	-O0	3655K	3625K	1.1%
combine.c	-O1	3199K	3531K	-11%
combine.c	-O2	3450K	3609K	-4.0%
combine.c	-O3	6245K	4086K	52%

Table 2: Peak GGC memory usage

Table 3: 64-bit SPECint 2000 -fnon-unit-at-a-time compared to -funit-at-a-time
Performance (relative speedup in percent, bigger is better):

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	avg
`-O2`	-0.89	1.22	0.72	0.00	0.42	0.35	3.51	4.84	-1.19	3.27	0.12	-4.36	0.24
`-O2 -m32`	-0.71	4.02	0.21	-0.19	-1.60	0.15	10.39	1.64	-1.82	-0.19	0.14	-0.61	0.86
`-O3`	-0.52	4.08	0.93	0.00	0.36	0.34	5.27	0.00	0.50	-0.50	-0.38	-4.27	0.11
`-O3 -m32`	-0.50	7.77	-1.93	0.00	-1.89	-0.71	6.36	0.96	0.26	1.52	-0.28	-1.53	0.61
`-O3` + profile	-1.78	3.91	0.19	0.00	-0.37	-0.35	3.84	3.91	-6.37	-1.61	0.49	-0.74	0.00
`-O3 -m32` + profile	-0.96	10.04	0.52	0.18	0.10	0.42	10.16	2.78	-0.89	-0.63	0.95	2.12	2.04

File size (relative increase of the size of stripped binaries in percent):

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
`-O2`	-20.42	-5.62	-2.08	0.00	-0.02	0.00	-8.58	-1.08	-0.10	-1.41	0.00	0.46	-2.63
`-O2 -m32`	-19.93	-2.66	-2.47	0.00	0.10	-0.03	-7.98	-0.89	-0.09	-0.87	0.00	-0.05	-2.44
`-O3`	-13.79	-1.47	5.14	0.00	3.68	4.17	-3.89	4.45	2.22	1.13	12.36	5.23	2.60
`-O3 -m32`	-12.72	3.62	5.48	0.00	4.33	5.28	-3.66	4.87	2.81	1.01	18.79	7.48	3.24
`-O3` + profile	-14.41	-1.33	5.18	0.00	2.35	4.12	-3.60	4.95	2.58	0.72	13.23	4.83	2.62
`-O4 -m32` + profile	-12.30	3.66	5.66	0.00	4.34	5.43	-3.68	5.21	2.99	1.02	18.29	5.79	3.29

Performance (relative speedup in percent, bigger is better):

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	apsi	total
`-O2`	0.00	0.14	0.00	0.00	-0.70	0.32	-0.13	0.00	0.00	0.00
`-O2 -m32`	-0.13	0.00	0.00	0.00	-1.36	1.48	0.72	0.00	0.00	0.17
`-O3`	0.00	0.00	0.00	0.17	-3.51	0.63	4.87	0.00	0.00	0.14
`-O3 -m32`	1.36	0.29	-0.18	0.00	4.67	1.89	3.75	0.00	0.00	1.02
`-O3` + profile feedback	0.11	0.43	0.00	0.00	3.35	1.92	1.74	0.00	0.00	0.86
`-O3 -m32` + profile feedback	0.00	0.00	0.18	0.00	7.36	2.80	3.01	0.00	0.00	1.19

File size (relative increase of the size of stripped binaries in percent):

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	apsi	total
`-O2`	0.00	0.00	0.00	0.00	-1.73	0.00	0.00	0.00	0.00	-0.47
`-O2 -m32`	0.00	0.00	0.00	0.00	-1.32	0.00	0.24	0.00	0.00	-0.35
`-O3`	0.00	0.00	0.00	0.00	-0.23	0.00	0.85	0.00	0.00	-0.06
`-O3 -m32`	0.00	0.00	0.00	0.00	-0.24	1.35	5.57	0.00	0.00	0.02
`-O3` + profile feedback	0.00	0.00	0.00	0.00	-0.24	0.00	0.00	0.00	0.00	-0.07
`-O3 -m32` + profile feedback	0.00	0.00	0.00	0.00	-0.21	1.43	5.16	0.00	0.00	0.03