Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Profile driven optimisations in GCC

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

Abstract

In recent 5 years GCC was redesigned to support profile driven optimization in almost all stages of optimization. We outline the approach for implementing the profiling support, the heuristics used to statically guess the profile and examine effectivity of individual optimizations. We also compare code quality produced by the static profile estimation relative to profile feedback and describe future plans on improving the profiling (such as value profiling on symbolic values to support devirtualization).

1 Introduction

To drive decisions of various optimisations (function inlining, loop optimisations, register allocation) it is useful to have information about execution time behaviour of the compiled code. By *control flow profile* we mean annotation of the internal program representation (control flow graph and call graph) by information about frequencies of executions of individual functions, basic blocks and edges.

There are several notions of profile commonly used to guide optimisations with different granularity. *Function profile* holds information about expected number of executions of individual functions, *basic block profile* holds information about expected number of executions of individual basic blocks, *edge profile* holds in addition information about number of executions (probabilities) of every edge in the control flow graph (CFG) and *path profile* holds information about execution of individual bounded size paths through the CFG.

In recent 5 years we modified GCC to use the edge profile that seems to be reasonable compromise across those choices—it is not more difficult to build than the basic block profile (except for low overhead profiling) while it is significantly easier to deal with than patch profiling scheme. Relative to basic block profiling it is significantly more useful to drive optimisations such as trace formation or basic block reordering (in fact some papers discussing basic block profiling approach also mention that edge probabilities needs to be guessed based on basic block profile, see [DEC]). For discussion of advantages of path profile over edge profile, see, for instance, [Path profile] or some of followup papers.

In addition to the control flow behaviour of the program, it is useful to have information about value histograms of individual variables manipulated by the program. We call such a profile *value profile*. This profile is useful to drive optimisations such as de-virtualisation, data prefetching or switch statement expansion.

The profile can be measured from actual program execution: program is compiled with instrumentation, trained on typical data set and later recompiled with taking the experimentally measured profile into account. This approach is very effective, but difficult to use. While several studies show that choice of train data set is not very difficult for most programs (the outcome of individual conditionals in programs seems to be mostly independent on choice of input data set), majority of users (especially in free software world) are not willing to complicate their build process.

Because of the mentioned problems it is useful to heave backup plan—static profile estimation. This technique estimates the edge profile based on the observation about common properties of the compiled programs without need for actual training. In limited sense (branch outcome guessing) it was first described by [Ball Larus]. Our implementation is mostly based on the followup paper [Static profile].

Rest of paper is organised as follows. In Section 2 we outline ways to construct and maintain the profile. In Section 3 we describe the more notable optimisations that take advantage of the profile. In Section 4 we present some SPEC2000 scores. In Section 5 we describe some issues arising from using profile guided optimisations in real world applications. Finally in Section 6 we summary our effort and describe some of plans for future enhancements we have.

2 Profile implementation

At our starting point GCC represented compiled functions as a doubly linked lists of instructions with almost no additional control flow data-structures preserved in between optimisation passes. While attempt was made to represent profile as a notes attached to instructions (branch probability notes for conditional jumps and execution count notes for the first instruction in “basic block”) it seemed most natural to represent profile as a part of control flow graph.

We took existing implementation of control flow graph used by liveness data-flow pass and added counts and frequencies to basic block data-structure. *count* is 64bit value representing number of executions of given block when profile feedback is available, *frequency* is 32bit value representing relative frequency within the function scaled to range 0 to 10000 and is available for static profile estimation as well as for profile feedback. Edges have similar notion of count, but the the relative information is recorded by *probability* of execution of the edge represented as integer in the range 0 to 10000. (It is important to have the integral frequencies kept in lower range as profile updating code sometimes compute up to third power of the values and we avoid use of floating point). We modified the existing passes to use and maintain the CFG up to date, so now (development `tree-profiling-branch` that plans to be merged into GCC 4.1) the CFG is build before inlining and is preserved up to the final stages of optimisation. Earlier released compilers (post GCC 3.2) maintained the profile only after the inlining.

Majority of profiling infrastructure and profile guided optimisers are trivial to implement and actually wast majority of effort was put into this project of updating compiler to preserve the CFG. We hope that these changes will pay back on their own in more maintainable and faster optimisation passes.

2.1 Profile feedback

2.1.1 Edge profiling

Edge profiling implementation predated our project and was originally contributed by James

E. Wilson in 1990. We based our implementation on this existing module, updated it to operate on CFG, extended the file format for more robustness and added value range profiling.

The basic idea of edge profiling is to insert counters on each edge of the CFG together with machinery to save them in runtime (our implementation in `libgcov` allows merging from multiple runs) and read back to compiler afterwards.

In GCC implementation the profile generation is optimised, so that not all edges in the CFG need instrumenting. First, the CFG is closed by adding fake edge from exit block to the entry block and extended to represent all places where execution might terminate (such as function calls and asm statements) by edges to exit block. To optimise the instrumentation we generate the CFG minimal span tree, only edges that are not on the span tree (plus the entry point) need instrumenting. From that information all other edge counts can be deduced via simple observation that sum of counts of incoming edges is equivalent to the sum of counts of outgoing edges for each basic block. To make instrumentation possible, all fake edges and edges we can not redirect (called *abnormal edges* comming from exception handling, indirect jumps and similar constructs) must be on the spanning tree. We also attempt to place critical edges on the spanning tree to reduce amount of newly created basic block.

Note that in some cases the abnormal edges might form an cycle making it impossible to include them all in the spanning tree. To prevent this we replace each such edge by edge from entry block to source block of replaced edge and edge from destination block of replaced edge to exit block. The resulting CFG has the property that spanning tree can always contain all the new abnormal edges even if the resulting profile can not be precisely transformed back to original one so actual counts of abnormal edges might end up wrong. Since the abnormal edges are unlikely optimised, this lack of information is probably not critical.

This algorithm results in significantly fewer instrumentation points (over 50% savings in average), but require the profile feedback to be read in carefuly since the errors might propagate across the CFG (further reduction might be possible by taking the estimated profile into account and construct maximal frequency spanning tree, but we didn't experimented with this idea yet). To prevent profile mismatches we do some basic CRC checks both in the runtime and at the time we load profile into compiler to ensure that the CFG used to guide instrumenting and current CFG match. We also ensure that the computed counts are not negative. Unfortunately even despite the CRC checking effort it is not unusual for user to get trapped by the corrupted profiles in some corner cases. Especially our implementation does not support threading and is not able to cope very well with constructors, destructors and dynamically loaded objects yet.

2.1.2 Value profiling

Value profiling was implemented by Zdeněk Dvořák and allows optimisers to point out values to profile by one of implemented value profiles (such as to discover most common value variable has, compute simple histogram or prove that it is mostly power of 2). At the present only very basic optimisations based on this infrastructure are implemented, but we hope to extend it soon to support speculative de-virtualisation to enhance C++ and Java inlining.

2.2 Profile estimation

When profile is not available, it is still possible to do reasonably good job by guessing the profile. Our implementation is mostly based on [Ball Larus] and [Static profile]. First series of heuristics are executed in attempt to predict direction of conditionals in the CFG. Because multiple heuristics might apply to given branch (and suggest different outcomes), the heuristics are noted in the CFG and later combined into single outcome. For reliable heuristics (see bellow) the outcome of first matching heuristics (in fixed priority order) is taken (such combination scheme is referred as *first match*), for unreliable heuristics *Dempster-Shaffer theory* (a statistics tool to combine hypothesis weighted by their reliabilities into single hypothesis) is used to produce overall outcome. See [Static profile] for details on Dempster-Shaffer theory.

When profile feedback is read, the information about success/failure of each heuristics might be dumped into debug file and tool `analyze_brprob` is available to measure hitrate of each hypothesis on the tested program. We regularly run such analysis on SPEC2000 integer benchmarks and put back the experimentally measured values into GCC to guide weights for combining the hypothesis.

While several followup papers to [Static profile] point out the Dempster Shaffer theory to be poor tool for combining the outcomes in this case (the mechanism incorrectly assumes full independence of the individual heuristics), we found this tool work pretty well (especially with the more reliable heuristics handled separately) and solve issues with discovering proper order of priorities for the "first match" scheme. [Ball Larus] took approach of experimentally testing every priority order for "first match" scheme instead that seems difficult to maintain as the number of different heuristics grow.

Following reliable heuristics are implemented (in the priority order for "first match" combination scheme):

loop iterations This heuristics attempts to analyse induction variables of each loop and figure out number of iterations. If this succeeds, the probability of loopback edge is computed. This heuristics match for typical `for` style loops with constant bounds on induction variable and similar constructs.

`__builtin_expect` As a GCC extension, `__builtin_expect` function call might be used to specify expected value of its argument. This information might be used to compute outcome of the branch. When this is possible, the expected direction is predicted with 95% probability.

noreturn call Code path leading to call that never returns (such as call to `exit` or `abort`) is unlikely executed. All basic blocks where some but not all exit edges leads to basic block postdominated by such a call gets the edges predicted as unlikely.

loop branch The loopback edge of loop is usually taken (since loops within time consuming programs are looping). For SPEC2000 average loop iterates roughly 5 times.

loop exit Edge exiting a loop is predicted as not taken. As a special cases basic blocks already predicted by loop branch heuristics are not considered for this heuristics.

This is very similar to loop branch heuristics, we just separated these two since most programmers tends to exit edges by the loop conditional rather than `break` statements placed in the middle of loop. This separation didn't seem to bring any useful value as can be seen from results in Section 4 however.

The following unreliable heuristics are implemented:

pointer Pointers are usually not `NULL`.

opcode positive Integral values in programs are usually positive (so outcome of conditionals comparing with 0 can be predicted).

opcode nonequal Two integral values in programs are not equal.

FP Opcode Two floating point values are usually not equal.

`goto` Heuristic predicting that `goto` statements are usually not executed. This was especially implemented to help people developing Linux kernel where `goto` statements are often used to get out of hot path. It seems to serve good job in other programs as well.

call Predict that a conditional where one but not all successor edges are postdominated by some specific call is not taken. The intuition is that conditionals are often used to protect error handling that is often represented by function call.

Adding function attribute to mark real error recovery functions (such as `perror`) might lead to significant improvement of this heuristics.

early return Predict that most functions returns by the last `return` statement in the source code of function. This is very fragile heuristics as it is common scheme to implement fast path checking in the beginning of function.

In the new implementation of our intermediate program representation, all the `return` statements are combined into single `return` at the end of program so this heuristics mark all the block leading to the return block early. This makes the heuristics even weaker than it used to be. Adding machinery to notice earlier `return` statements might be possible, but author is unsure if this heuristics is worth the effort at all.

constant return Returns with constant values are often used to represent error states. This heuristic predict return of nonzero constant to be unlikely.

negative return Predict return of negative constant to be unlikely (negative values are even more likely to represent error state)

`NULL` return Predict `return NULL;` to be unlikely

Following predictors are not functional at a time of writing paper because of unfinished effort to updated them for tree-SSA framework.

`continue` A loop formed by C `continue` statement is usually not looping. Hitrate was about 56% demonstrating that average loop formed by `continue` iterates usually just twice, while normal loop 5 times. As a special case we disabled loop branch heuristics when continue heuristics matched. We plan to re-implement this soon.

Loop header It is common to transform `while` into `do-while` style loops by duplicating the loop conditional. The first execution of loop conditional is usually (it decides whether a loop will iterate at all or not) and the heuristics claims that the loop will usually execute. The hitrate of this heuristic used to be roughly 64% but in the new implementation we duplicate the loop test after constructing the estimated profile and this heuristic is thus no longer in use.

Note that our implementation was different from [Ball Larus] one in a way that we marked only the really duplicated tests and this improved reliability of this predictor.

Once the probabilities of individual edges are computed, the estimated profile is constructed by propagating the information across non-loop regions and computing expected number of iterations of natural loops by same algorithm as in [Static profile]. This algorithm runs into interesting side cases, where function having CFG of very deep tree might run into exponentially small frequencies that still needs to sum to to entry block frequency in exit, while CFG with very deep loop nest can run into exponentially large frequencies. This is a proof of lack of sense of reality in the profile but also problem for implementation since algorithm runs easily out of range of integers. Our initial implementation used floating point but it resulted in different profile being constructed on different underlying FP units so we replaced it with our software emulator that turned out to be bottleneck (for very deep loop nests, the algorithm is quadratic) and now we use custom floating point implementation.

The branch prediction algorithm is run in limited sense even when profile feedback is available. For basic blocks with count of 0 the edge probabilities are guessed and for functions never executed in the train run the estimated frequencies are constructed based on these probabilities (on executed functions, the basic block counts are simply rescaled to frequencies and edge counts are used to compute probabilities).

2.3 Maintaining profile up to date

Touchy issue is the need to keep profile intact in CFG across the whole compilation process. While for the CFG itself it is easy to implement sanity checking tool and hide most of actual transformations in simple abstraction (such as function redirecting the edge in CFG at the same time as updating underlying IL), the profile might easily become damaged even by properly performed transformations. The basic problem is that each code duplication transformations might result in two different copies of original region of CFG with different profiles while the lack of path sensitive profile information in compiler generally only allow to copy distribute the profile evenly across the copies. Later constant propagation on conditionals might then prove edge with nonzero believed probability impossible resulting in conflicts. This makes automatic verification difficult (if not impossible at all). In a combination with the fact that most of profile updating requires high level knowledge about the transformation the optimisation is performing (so it is difficult to hide in basic CFG manipulation abstraction) and the fact that many of GCC developers are still unaware of the profile existence, it is common for profile to break. At the moment only sanity checking is done while dumping the intermediate representation into debug files, but most of GCC developers seems to ignore the mismatches found.

3 Optimisations implemented

3.1 Register allocation

GCC use simple priority driven register allocation scheme. In order to compute priorities, the basic block frequencies are now taken into account. This replaced original heuristics that estimated every loop to iterate three times.

3.2 Hot/cold partitioning

Many of optimisations trading code size for performance are now conditioned by predicates `maybe_hot_p`, `probably_cold_p` and `probably_never_executed`. When profile feedback is available, the basic block is considered as maybe hot when its count exceeds maximal count across all compilation units (computed by runtime saving the profile) divided by count fraction parameter defaulting 1000. Basic blocks not satisfying probably hot predicate are considered maybe cold and basic blocks having counts lower then half of training runs are considered never executed.

When profile feedback is not available the predicates must be computed on per-function basis and are mostly noop. We consider all blocks maybe hot except those having basic block frequency smaller than maximal frequency divided by frequency fraction parameter (defaulting to 1000). Analogously for maybe cold predicate. Blocks are never considered as probably never executed. Because the estimated profiles tends to be flat, it is likely that most functions will consists only of maybe hot blocks.

Predicate `maybe_hot_p` is used by majority of code growing optimisation. It is used by loop unrolling and unswitching to disable transformation on uninteresting loops, basic block reordering to disable basic block duplication, inliner to avoid code growing inlines in cold calls, crossjumping to avoid merging of basic blocks with different outcomes and few extra transformations. Probably never executed is currently in use in basic block reordering algorithm to factor out uninteresting portions of functions and by pass inserting code alignments.

3.3 Function partitioning

To reduce code cache footprint and average call distance, we split functions into separate sections on ELF system based on their execution frequency. `.text.hot` section contains functions with at least one hot basic block. `.text.unlikely_executed` contains functions where all blocks are unlikely executed. Rest of functions go into `.text`.

Recently function body splitting was implemented (so large function body can be divided into those sections when it contains hot and cold portions) but this feature is not fully functional at a time of writing this paper and is disabled in the benchmarks. It would be also nice to implement full function reordering algorithm (that reorders functions in a way so common calls are in short distance) but that would require closer cooperation with the linker or whole program optimisation.

3.4 Tracer

Tracer (also known as tail duplication or superblock formation) is a pass duplicating basic block across common paths in the code to avoid "join edges" (places where the common path meets with some other path in CFG causing dataflow based optimisation going into conservative assumption). Our implementation is based on [DEC] with few additional parameters to limit code growth.

The pass works by identification of *traces* i.e. sequences of basic blocks often executed in a row. Then the blocks are visited in trace order, starting with the second block. If a block has more than one predecessor the block is duplicated (with all outgoing edges) and the edge from previous block of the trace is redirected to the duplicate. This is repeated until every basic block on the trace is visited. As a result the trace is transformed into a *superblock* (i.e. sequence of basic blocks with no incoming edges except for the first basic block in sequence).

The usual algorithm for trace formation starts in some frequent basic block (seed) and continues by the most frequent predecessor as long as the previously visited basic block is the most frequent successor of selected predecessor. Once the walk terminates it is performed symmetrically from the seed to the most frequent successor. Our implementation works by repeatedly choosing the most frequent basic block not yet visited by trace formation algorithm as a seed, forming a traces and performing tail duplication until some of limiting parameters are met.

Tracer is one of algorithms not working very well on estimated profiles, since it care about outcome of non-loop edges that gets often misspredicted. Little special care however makes the algorithm profitable on estimated profile too (as observed earlier in [Superblock], our methods are different thought).

We limit the code growth caused by algorithm by several parameters. *Minimal probability of successor* limits the probability of edge that still can join two basic blocks in the trace. This value needs to be set differently for estimated profile where one hardly finds edges with very high probability (while in real profiles this is very common). We use 80% probability for real profile and 50% probability for estimated profiles. *Dynamic coverage* limits amount of basic blocks ever considered for superblock formation. The algorithm is run as long as the time spent in the visited blocks does not exceed this parameter. This time for real profile one needs higher value (95%) while for estimated profile we use 75% as code size growth gets too considerable otherwise. This seems to be explained by fact that estimated profiles are more “flat” (the difference in between coldest and hottest basic block is lower than in real profile). Last parameter (*max code growth*) cuts the duplication process once the function body grows by given amount.

In GCC implementation the trace formation is run in the RTL part of backend after all high-level and mid-level optimisations are performed. This is done basically for historical reasons and we plan to re-implement the tracer as part of SSA optimisation queue. We however need to experiment with this carefully as running tracer damages dominator relationship and might confuse high-level optimisations.

The benefits of tracing come mostly from improved optimisation possibilities for later passes and from reduced amount of taken branches executed in the final code. Our implementation is targeted specially for the first goal. The code paths where no optimisation took place should be later re-combined by tail merging pass done after register allocation (due to limited scope of analysis done here this is not always the case) and we re-run similar algorithm in the basic block reordering pass to track the second case.

3.5 Loop unrolling and peeling

Profile provide information about expected number of iterations of each loop in program as well as about importance of each loop. This is very useful to drive loop optimisations in general. Our implementation of unroller will not unroll loops in cold regions of program. When profile feedback is not available loops where number of iterations is not known are not unrolled by default. When feedback is available, the loops iterating more than twice the number of copies unroller is considering to make are unrolled and other loops are peeled.

The unroller is currently enabled by default only when profile feedback is available as otherwise it is considered too costly code size wise for common use.

3.6 Inliner

Our inliner implementation use simple bin packing algorithm. There are several limitations given by user (maximal estimated size of inline candidates, maximal growth of whole compilation unit due to inlining, large function threshold and maximal growth of large functions). Given these limitations, the inliner is attempting to maximise number of calls inlined. When profile feedback is available we compute priorities based on number of execution of each edge in the call-graph and the expected growth of caller function body after inlining is performed (so the priority represents benefits to costs ratio in our simplified model). Then simple greedy heuristics attempt to mark inlinable edges inline in the priority order dynamically updating the costs and priority queue as the number of call sites of functions change. When profile feedback is not available we optimize for amount of different functions all callers—so the priority is driven by overall unit growth after inlining into all callers of given function.

In the benchmarks we included results with profile feedback available but inliner modified to ignore them (so it would be still optimizing for number of different functions inlined). These results are referred as “static inliner”.

Inlining is certainly one of the most important profile driven optimisations because its benefits can be very high (some C++ programs can be sped up several times) and the costs can go up as well. Sadly our estimated profiles don’t seem to be very useful for driving inliner yet, because the profile is not propagated inter-procedurally. [Static profile] suggest that adding inter-procedural profile propagation pass might be fruitful for whole program optimisation.

3.7 Basic block reordering

Basic block reordering (branch alignment) is a pass changing order of basic blocks with a goal to minimize amount of taken branches. This can be done by simple DFS search that visits the edges in probability order, but it is far from optimal. The optimal sollution can be reduced to travelling salesman problem [TSP] that is hard to solve so we use easier approach (software trace cache, see [STC] for full description of the algorithm) that is in nature similar to trace formation: first traces are identified and then ordered in sequence with limited amount of basic block duplication in most profitable cases.

3.7.1 Value profile transformations

At the moment just very simple transformations are implemented. We have pass converting:

```
a=b/c;
```

into:

```
if (c==common_value_of_c)
  a=b/common_value_of_c;
else
  a=b/c;
```

or

```
if (c is power of 2)
  a=b << log(c);
else
  a=b/c;
```

similarly for modulo operations where we additionally check for the case where modulo is noop.

While this optimisation is good trick for extra SPEC points we are unsure it serve very good value in practice (even though GCC itself is sped up by this transformation measurably in its growing hash tables functions). We currently work on speculative de-virtualisation that seems more important use of this framework.

We also implemented a speculative prefetching. We look for read or write of `mem[`*address*`]`, where the value of *address* changes usually by a non-zero constant C between the following accesses to the computation. We then add a prefetch of $address + C$ before the load/store. This handles prefetching of several interesting cases in addition to a simple prefetching for addresses that are induction variables, e.g. linked lists allocated sequentially (even in case they are processed recursively).

For majority of modern CPUs this optimisation is not particularly interesting since the hardware prefetching mechanism is in charge. This transformation is currently disabled because we didn't updated it to new tree-SSA framework (yet).

4 Experimental results

In this section we present some results of SPEC2000 benchmark on AMD Opteron CPU to give an idea of effectivity of various optimisations. All runs are done with standard optimisation settings and whole program compilation enabled (`-O2 -fwhole-program --combine`).

Tables 1, 2, and 3 refer to outcomes of individual branch prediction heuristics described in Section 2.2. *Branches* column list number of branches in compiled code the given heuristics match on, *coverage* represent number of branches executed predicted by given heuristics, *hitrate* represents the probability that outcome of executed branch predicted by the heuristics will actually be as predicted and *max hitrate* represents maximal such hitrate possible (i.e. the predictability of the branch by profile feedback). So good heuristics should have coverage and hitrate as high as possible. If hitrate is lower than 50%, the heuristics are wrong in most cases on given benchmark.

Individual heuristics are presented first, *first match* represents the combined outcome of all reliable heuristics, *DS theory* represents combined outcome of all unreliable heuristics, *no prediction* represents all branches not predicted at all. Finally *combined* represents outcome of all the heuristics together.

As can be seen, the usability of individual heuristics may vary depending on programming style and nature of problem it solves (SPECfp control flow behaviour is much more predictable overall as most of time is spent by loop branches) but overall the heuristics combination scheme derive useful results even for non-SPECint programs our prediction mechanism is tuned for. The combined hitrate of 75% for SPECint seem to be very good result compared to other papers published on the topic. Our main advantage over [Static profile] implementation seems to be availability of higher level information about the source program since our predictors works early in the optimization queue, while [Static profile] analyse already optimised object code.

Tables 4 and 7 list effectivity of most important optimisations we implemented when guided by static profile estimation. Tables 5 and 6 list results with profile feedback. The first line in each table represents the same baseline run done twice to give a idea of noise in the benchmarks. It is followed by optimisations enabled in the standard optimisations settings in order of effectivity. Optimisations not enabled by default but available as additional command line options come next (note that the standard settings differ when profile feedback is available as the inliner, superblock formation and loop transformations are enabled by default). Summary of effectivity of all the profile guided optimisations come last.

Each benchmark was repeated 3 times and median of the runs was used. Since it is not possible to cover the interferences in between passes, the benchmark always compare the results with the pass of interest disabled relative to run with the pass enabled. All other passes are enabled/disabled same way as with default settings. SPEC2000 benchmarks provide two different datasets one for train run and one for actual benchmark so it should come close to real world scenario.

As can be seen, most of optimisations are already effective with statically estimated profile, but the additional gains from actual profile are still important. Some of optimisations are less effective than in [DEC], but it seems to be explained by lower sensitivity of AMD Athlon to compiler optimisations than traditional RISC architectures. The higher effectivity of register allocator seems to support this hypothesis.

For runs with profile feedback, *profile estimation* refers to enabling profile estimation in addition to profile feedback for portions of programs not covered by train runs. *profile feedback* compare performance with estimated profile relative to performance with profile feedback and estimation enabled. *overall* compare performance with no profile compared to performance with profile feedback and estimation enabled. It is somewhat surprising that the profile estimation is quite effective even in addition to the actual profile feedback.

Author finds it somewhat disappointing however that the code size is increased by the profile feedback despite all the benefits from hot/cold partitioning. This seems to be combined result of enabling aggressive code growing optimisations by default. Obviously there is room for improvements in this area because majority of other compilers produce smaller code when profile feedback is available.

5 Real world usage

While the branch prediction mechanism is completely transparent to the end user and seems to be widely accepted (there are almost no problem reports related to branch guessing in bugzilla and only very few projects disable profile driven optimisation explicitely. In fact author is only aware of linux kernel that disable basic block reordering pass and rely on `goto` statements instead), the profile feedback feature has not been very widely accepted by free software community a the present.

This situation is understandable since for free software the simplicity of build process is very important. On the other hand it is unfortunate too as profile feedback brings considerable improvements both in performance and memory footprint. Author hope that the feature will become more popular in the future as the quality of implementation improve. The situation would be also greatly helped if common tools, such as automake had basic support to specify train run and produced profile feedback ready makefiles.

GCC itself is able to build with `profiledbootstrap` setting that use its own runtime libraries as train run. The speedups on x86 architecture (AMD Athlon) are roughly 7% building `combine.c` GCC module with optimization, 10% building without optimisation and 15% building simple hello world program (an average of 100 runs).

Several core interpreter packages (bash, awk, sed, etc...) in SUSE GNU/Linux distribution are also built with profiling via simple trick:

```
CFLAGS="-O2 -fprofile-generate"\
 make
make check
make clean
CFLAGS="-O2 -fprofile-use" make
```

These packages are specially easy targets for feedback based optimizations and the benefits are slightly over 10% on execution of common configure scripts from disc cache.

Author is also aware of at least one proprietary database vendor who used the profile feedback feature in GCC.

6 Conclusion and future plans

The overall speedup of 6% on SPECint (12% with profile feedback) makes the profile guided optimisations one of the most effective improvements to GCC optimisers in recent few years. We found the static profile estimation a valuable and almost drop-in replacement for real profile for number of profile guided optimisations. Work on CFG transparency across optimisation passes should have improved the maintainability of GCC sources too. This makes us believe that our work seems relatively effective despite the long term effort needed and the fact that profile feedback feature might not gain very big user base. We also hope this paper to be useful as perhaps the first study of effectivity of static branch prediction algorithms in real world compiler.

We might have captured most of the lowest hanging fruits but still in addition to possible improvements discussed earlier we see many of areas that needs further work. In near future we plan to add devirtualisation support and finish updating the infrastructure for tree-SSA optimisers. We also hope to improve user friendliness and usability of profiling mechanism (add more safety checks, thread support). The `gcov` utility to annotate source code by number of executions of individual basic blocks/edges can be greatly extended too. Several compiler vendors are shipping popular tools for source level coverage to detect dead code in programs but our `gcov` utility don't seem to be that popular yet, perhaps because it lacks number of features, especially good summary information and not everyone is ready to employ his favorite scripting language.

The static profile estimation methods can always be improved too. GCC recently gained a value range propagation pass that can be useful to predict non-loop branches (see [VRP]). Several other ideas are discussed in number of followup papers to [Static profile].

There is no interface for passing profile information to machine descriptions. This can be useful to drive the instruction choice in expanders, splitters and output templates so the choice in between code size and optimised performance instruction choice can be done on local basis.

Interesting possibility for future development is to employ the low overhead profilers (such as `oprofile`). Even though those profilers are not providing as much granuality as we might want, they should be useful to drive the inlining decision and hot/cold partitioning (the most important passes we can not drive by static profile). Combined with profile estimation doing pretty good job on local basis, the resulting code may get pretty close to one produced by actual edge profile. Low granuality of the profile might also make it possible for developers to measure the profile themselves and simply ship the relative function weights together with the program source code.

Finally many of optimisers implemented (profile guided inliner in particular) would probably benefit from additional tuning as so far majority of effort went into getting the infrastructure in place and little was put into actual tweaking of individual passes.

7 Credits

Over the years, many of GCC developers contributed to the project. Steven Bosscher helped to cleanup CFG inliner and CFG transparent RTL expansion patch. Zdeněk Dvořák wrote value profiling pass and new loop unroller. Stuart Hastings implemented basis of CFG transparent inliner. Richard Henderson reviewed majority of the patches. Jan Hubička implemented most of CFG transparency changes, profile representation, tracer and is co-maintaining the profiling code. Dale Johannesen updated value profiling for tree-ssa infrastructure and co-developed CFG transparent inliner. Pavel Nejedlý improved file format used by profiling runtime. Nathan Sidwell is co-maintaining profiling code and implemented profile merging and number of improvements into the runtime, on-disc file format and `gcov` utility. Caroline Tice implemented function splitting. Original edge profiling was implemented by James E. Wilson, later updated by Bob Manson. Josef Zlomek implemented software trace cache algorithm.

References

[Static profile] Y. Wu and J.R. Larus, *Static branch frequency and program profile analysis*, Proceedings of the 27th International Symposium on Microarchitecture (1994), p. 1–11.

[Path profile] T. Ball, J.R. Larus., *Efficient path profiling*, Proceedings of the 27th International Symposium on Microarchitecture (1996), p. 46–57.

[Superblock] R.E. Hank, S.A. Mahlke, R.A. Bringmann, J.C. Gyllenhaal, W.W. Hwu, *Superblock Formation Using Static Program Analysis*, International Symposium on Microarchitecture (1993), p. 247–256.

[Ball Larus] T. Ball and J.R. Larus, *Branch Prediction For Free*, Proceedings of the 27th International Symposium on Language Design and Implementation (1993), p. 300–313.

[DEC] *Design and Analysis of Profile-Based Optimization in Compaq's Compilation Tools for Alpha*, Journal of Instruction-Level Parallelism 3 (2000), p. 1–25.

[STC] A. Ramirez, J.L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, *Software trace cache*, Proc. 13th Intl. Conf. on Supercomputing (1999), p. 119–126.

[TSP] Cliff Young, David S. Johnson, David R. Karger, Michael D. Smith, *Near-optimal Intraprocedural Branch Alignment*, ACM 1997.

[VRP] J.R.C. Patterson, *Accurate Static Branch Prediction by Value Range Propagation*, 1995.

Table 1: Branch prediction heuristics behaviour on SPEC2000 integer train run

Heuristics	branches	(rel)	hitrate	maximal hitrate	coverage	(rel)
loop branch	13883	8.4%	91.35%	92.95%	17352864750	23.9%
loop exit	11212	6.8%	91.64%	94.75%	12891913888	17.7%
call	51548	31.2%	71.90%	93.46%	9807380439	13.5%
opcode values nonequal	31239	18.9%	70.98%	88.60%	8578270991	11.8%
early return	14386	8.7%	45.98%	91.65%	4519214662	6.2%
pointer	19467	11.8%	83.35%	92.61%	3438347274	4.7%
opcode values positive	6348	3.8%	73.67%	85.01%	2594503596	3.6%
loop iterations	1975	1.2%	94.11%	94.11%	1528650186	2.1%
noreturn call	1740	1.1%	100.01%	100.01%	408485170	0.6%
negative return	2010	1.2%	77.30%	79.14%	305719479	0.4%
null return	1336	0.8%	91.82%	94.51%	239072867	0.3%
`__builtin_expect`	148	0.1%	88.81%	88.81%	93907054	0.1%
DS theory	93449	56.5%	72.89%	89.69%	23017931222	31.7%
first match	27821	16.8%	91.64%	93.74%	31750839958	43.7%
no prediction	44078	26.7%	51.07%	85.61%	17866106188	24.6%
combined	165348	100.0%	75.72%	90.46%	72634877368	100.0%

Table 2: Branch prediction heuristics behaviour on SPEC2000 fp train run

Heuristics	branches	(rel)	hitrate	maximal hitrate	coverage	(rel)
loop branch	1421	14.4%	95.73%	95.73%	4627361472	50.6%
loop exit	339	3.4%	95.82%	95.82%	1697791223	18.6%
call	2952	30.0%	67.48%	94.23%	822816942	9.0%
opcode values nonequal	784	8.0%	22.11%	94.42%	590441990	6.5%
early return	1295	13.2%	22.10%	92.16%	210748393	2.3%
pointer	1352	13.7%	99.24%	99.96%	164797817	1.8%
opcode values positive	1025	10.4%	78.33%	80.41%	108902513	1.2%
loop iterations	242	2.5%	94.50%	94.50%	83631750	0.9%
null return	45	0.5%	49.60%	99.99%	24961335	0.3%
noreturn call	112	1.1%	99.99%	100.00%	1352502	0.0%
negative return	27	0.3%	50.00%	100.00%	12	0.0%
`__builtin_expect`	1	0.0%	0%	0%	0	0.0%
DS theory	4943	50.2%	56.50%	93.00%	1261323000	13.8%
first match	2101	21.4%	95.73%	95.73%	6400679358	70.0%
no prediction	2794	28.4%	48.77%	90.21%	1478703393	16.2%
combined	9838	100.0%	82.72%	94.46%	9140705751	100.0%

Table 3: Branch prediction heuristics behaviour on GCC profiledbootstrap

Heuristics	branches	(rel)	hitrate	maximal hitrate	coverage	(rel)
loop exit	11000	7.7%	86.28%	92.20%	482487861	20.2%
opcode values nonequal	34145	24.0%	67.58%	89.78%	436206375	18.3%
pointer (on trees)	21970	15.4%	63.15%	91.27%	351688646	14.8%
noreturn call	21523	15.1%	99.99%	100.00%	335263906	14.1%
call	37695	26.5%	69.24%	89.90%	333274963	14.0%
early return (on trees)	17225	12.1%	55.38%	89.89%	238913240	10.0%
loop branch	8093	5.7%	84.82%	85.94%	187124950	7.8%
loop iterations	381	0.3%	87.27%	87.27%	65497338	2.7%
opcode values positive	2810	2.0%	67.47%	88.23%	53403249	2.2%
null return	1771	1.2%	68.53%	83.50%	44592831	1.9%
negative return	9486	6.7%	17.32%	94.87%	17173041	0.7%
`__builtin_expect`	24	0.0%	97.53%	99.61%	623140	0.0%
DS theory	82062	57.6%	65.38%	89.99%	1083353422	45.5%
first match	38990	27.4%	89.31%	92.49%	876724378	36.8%
no prediction	21294	15.0%	41.18%	86.26%	422370441	17.7%
combined	142346	100.0%	69.90%	90.25%	2382448241	100.0%

Table 4: 64-bit SPECint 2000 without profile feedback (AMD Opteron)
Performance (relative speedups in percent):

	gzip	vpr	gcc	mcf	crafty	parser	perl	gap	vortex	bzip2	twolf	avg
	0.00	–0.43	–0.43	0.24	0.21	0.00	–0.11	0.00	0.00	0.00	0.46	0.12
register allocation	0.00	1.56	1.17	0.00	2.05	–0.73	0.00	4.94	–2.20	–1.32	5.76	1.38
block reordering	0.25	0.42	0.42	–0.25	1.68	–0.37	0.93	1.15	0.72	0.80	–0.35	0.49
hot/cold blocks	–0.13	0.42	1.06	2.24	2.05	0.36	0.10	–0.13	0.00	–0.40	0.23	0.49
superblock formation	–0.26	1.54	0.10	0.24	–1.52	0.91	1.75	0.63	4.22	3.07	1.04	1.11
static inliner	3.32	3.64	–0.32	–2.35	1.01	6.95	–12.16	0.38	9.54	0.78	–2.71	0.24
loop unroll/peel	0.76	–1.40	–0.43	–4.67	–1.73	–1.47	2.67	0.38	0.09	0.00	–0.94	–0.74
profile estimation	–0.26	3.33	8.60	–0.25	7.01	3.99	11.26	12.78	10.75	5.31	5.89	6.17

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

options	gzip	vpr	gcc	mcf	crafty	parser	perl	gap	vortex	bzip2	twolf	total
hot/cold blocks	0.00	0.00	–0.01	0.00	0.00	0.00	0.00	–0.03	0.00	0.00	0.00	–0.01
register allocation	0.00	0.01	0.46	0.00	0.00	0.00	0.23	0.00	0.21	0.00	0.43	0.26
block reordering	3.79	1.95	2.93	0.00	3.17	0.00	3.44	2.54	3.82	0.00	2.73	2.90
superblock formation	0.39	2.03	2.06	0.00	1.70	2.11	2.97	3.48	2.28	–0.05	2.28	2.35
static inliner	5.76	2.69	8.43	2.78	3.41	22.37	7.46	6.63	1.86	12.71	5.42	6.96
loop unroll/peel	46.45	28.70	11.41	27.42	16.03	14.93	14.45	45.12	1.77	31.65	38.36	17.81
profile estimation	3.94	1.87	3.02	–0.11	1.65	1.95	3.58	2.29	4.96	6.75	2.31	3.16

Table 5: 64-bit SPECint 2000 with profile feedback (AMD Opteron)
Performance (relative speedups in percent):

	gzip	vpr	gcc	mcf	crafty	parser	perl	gap	vortex	bzip2	twolf	avg
	0.00	0.39	0.19	0.00	0.00	–0.15	–0.10	–0.25	0.08	0.00	–0.36	0.00
block reordering	3.16	1.45	2.10	–0.25	0.43	1.16	3.69	3.50	2.20	1.70	1.08	1.78
value profiling	0.83	–0.26	0.00	–0.49	0.07	16.05	0.56	0.24	0.24	0.51	–0.36	1.42
profiled driven inliner	3.79	5.63	0.59	–0.48	0.58	5.98	1.42	1.97	4.68	0.51	–3.38	1.42
loop unroll/peel	2.16	0.13	3.97	–0.25	3.07	1.61	3.90	0.98	8.26	0.12	–3.90	1.30
superblock formation	1.31	4.06	0.29	–0.25	–0.73	0.14	0.28	1.35	0.24	0.38	0.60	0.70
function partitioning	0.11	0.65	–0.20	–1.20	0.58	0.28	2.10	0.97	–0.50	0.12	2.06	0.58
register allocation	1.31	–0.14	1.09	–0.25	0.51	1.16	1.13	0.24	–2.75	0.64	1.44	0.47
hot/cold blocks	–0.12	0.65	–0.20	–0.25	–0.30	0.14	3.49	0.00	0.00	0.12	–3.12	–0.24
static inliner	2.81	4.94	–0.20	–0.24	–0.88	6.74	0.47	0.73	3.90	–0.26	–5.13	0.59
profile estimation	2.66	1.86	2.83	–0.49	2.69	1.60	2.30	1.22	4.69	0.91	1.69	1.90
profile feedback	8.71	6.22	7.15	0.97	0.80	25.72	8.12	4.17	3.87	0.64	–3.79	4.65
overall	8.16	11.46	16.85	0.72	5.85	31.93	22.41	18.39	20.01	8.53	2.94	12.22

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

options	gzip	vpr	gcc	mcf	crafty	parser	perl	gap	vortex	bzip2	twolf	total
hot/cold blocks	–15.65	–14.62	–7.67	0.00	–11.30	–6.17	–11.04	–24.43	–10.79	–14.44	–19.56	–12.23
function partitioning	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.02	0.03	0.00	0.00	0.01
block reordering	0.00	–0.14	0.20	0.00	0.00	0.00	0.00	–0.23	–0.05	0.00	–0.13	0.02
value profiling	0.00	0.00	0.00	0.00	0.00	–0.07	0.00	0.16	0.17	0.00	0.55	0.07
register allocation	0.00	0.05	0.19	0.00	0.00	0.00	0.00	–0.08	0.22	0.00	0.14	0.10
superblock formation	–0.08	0.07	1.20	0.00	0.00	0.00	1.01	0.97	1.90	0.00	0.86	1.02
inliner	1.77	1.78	2.32	12.04	0.00	15.50	0.53	1.50	0.31	5.95	0.60	2.06
loop unroll/peel	11.31	5.73	3.36	23.43	5.85	13.83	1.08	4.95	0.67	12.66	2.89	3.76
static inliner	6.13	3.64	9.54	12.04	3.19	29.81	8.13	7.79	1.88	17.91	5.32	8.12
profile estimation	–0.04	–0.15	–0.02	0.00	–0.04	0.84	–0.04	0.00	0.15	0.00	–0.17	0.03
profile feedback	10.57	4.79	2.07	27.53	5.06	25.17	–3.05	0.14	–3.82	12.66	0.68	1.53
overalll	15.38	8.93	7.34	27.39	8.62	30.32	3.42	6.01	3.25	20.22	5.36	7.20

Table 6: 64-bit SPECfp 2000 without profile feedback (AMD Opteron)
Performance (relative speedups in percent):

options	wupwise	swim	mgrid	applu	mesa	art	equake	facerec	ammp	lucas	fma3d	sixtrack	apsi	total
	–3.89	0.72	0.00	0.16	–2.25	–1.07	2.81	–0.79	–0.12	0.00	1.45	0.96	0.78	0.00
register allocation	6.04	0.72	–1.84	3.46	2.62	–0.64	1.11	1.44	–1.50	0.57	–1.30	–1.67	4.67	1.26
hot/cold blocks	0.00	0.14	0.00	0.00	0.00	1.08	–0.13	0.00	–0.24	0.23	–0.44	0.24	0.26	0.13
block reordering	1.22	0.14	0.18	0.00	–1.91	–0.32	–2.04	0.15	0.11	0.23	–1.02	–6.14	0.39	–0.69
loop unroll/peel	2.30	–1.44	–2.07	5.51	–2.77	0.63	–4.52	5.98	6.06	–0.12	2.89	7.03	5.92	2.21
static inliner	0.00	0.28	0.00	0.16	3.37	–2.72	–2.43	–0.16	0.11	–0.12	0.00	0.00	0.26	0.00
superblock formation	–4.00	0.28	0.00	0.00	–2.35	–2.64	3.06	–0.79	–0.12	–0.12	1.45	1.21	0.78	–0.14
profile estimation	7.15	1.01	7.87	5.47	10.62	1.52	3.80	2.25	4.26	3.20	–0.30	2.99	4.95	4.18

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

options	wupwise	swim	mgrid	applu	mesa	art	equake	facerec	ammp	lucas	fma3d	sixtrack	apsi	total
hot/cold blocks	0.01	0.03	0.02	0.01	0.00	0.00	0.00	0.00	0.00	–0.07	0.00	0.00	0.00	0.00
register allocation	0.00	0.27	0.00	0.18	0.00	0.00	0.00	0.14	0.01	0.19	0.00	0.28	0.00	0.09
block reordering	0.00	0.40	0.00	0.18	0.00	0.00	0.00	0.55	2.23	0.19	0.72	1.10	0.73	0.71
static inliner	0.00	0.00	0.00	0.00	4.98	5.20	2.79	0.00	2.98	0.00	0.00	0.00	0.00	1.20
superblock formation	1.13	0.13	0.22	–0.05	2.85	0.00	0.00	4.15	4.36	0.35	2.15	0.38	2.94	1.81
loop unroll/peel	25.52	25.78	35.38	41.21	38.23	41.24	32.47	29.53	37.10	6.15	31.67	19.44	37.13	29.52
profile estimation	–0.07	0.54	0.11	0.18	2.87	0.00	–0.09	0.29	2.12	–0.24	0.64	0.77	0.20	1.10

Table 7: 64-bit SPECfp 2000 with profile feedback (AMD Opteron)

Performance (relative speedups in percent):

options	wupwise	swim	mgrid	applu	mesa	art	equake	facerec	ammp	lucas	fma3d	sixtrack	apsi	total
	–0.11	0.28	–1.73	–0.76	0.00	0.00	–0.37	–0.15	–0.34	–0.12	0.00	–0.23	–0.27	–0.27
loop unroll/peel	13.31	0.14	–1.54	8.77	0.36	0.75	–3.68	4.41	2.40	0.00	2.15	7.17	1.36	2.60
block reordering	4.07	0.43	0.19	0.30	10.87	–1.58	–0.13	0.58	1.12	0.22	0.28	0.22	0.53	1.21
superblock formation	7.99	0.14	–1.73	–0.76	4.60	0.00	0.00	0.00	–0.89	3.77	–0.29	1.35	0.81	1.08
function partitioning	0.21	0.86	0.19	0.00	10.53	0.64	–0.13	0.14	0.11	0.11	–0.15	0.22	0.53	0.94
value profiling	3.84	–0.15	–1.54	–1.06	8.77	–0.85	–1.22	–0.44	–0.67	0.11	–0.43	–0.23	0.40	0.53
profile driven inliner	0.00	0.14	0.19	0.30	0.00	0.95	0.85	–0.15	–0.23	0.00	0.00	–0.23	–0.14	0.13
register allocation	1.65	–0.15	–5.94	1.37	–1.72	–0.95	0.12	0.29	–0.67	–1.57	0.42	0.67	1.91	–0.14
hot/cold blocks	2.22	1.15	2.14	1.53	2.24	1.40	–4.90	0.73	0.11	–0.46	0.56	0.22	–5.21	–0.27
static inliner	0.00	0.00	0.00	–0.16	–0.82	0.21	0.24	–0.58	–0.23	0.00	–0.43	0.00	–0.14	–0.14
profile estimation	0.65	0.86	–0.20	0.30	1.11	–0.43	–0.13	–0.15	–0.23	0.22	1.13	0.22	1.91	0.53
profile feedback	16.03	0.14	–3.75	9.86	14.00	1.62	–3.33	9.06	4.79	1.26	2.15	7.43	–3.13	3.60
overall	19.50	1.89	3.83	16.07	23.27	2.06	3.17	10.64	9.13	4.51	3.34	11.72	2.47	7.93

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

options	wupwise	swim	mgrid	applu	mesa	art	equake	facerec	ammp	lucas	fma3d	sixtrack	apsi	total
hot/cold blocks	–8.84	–3.91	–12.80	–2.22	–25.78	–8.52	–7.51	–17.14	–24.25	–10.96	–11.97	–19.87	–18.30	–17.62
block reordering	4.38	–0.23	–0.20	–0.07	–0.57	–0.16	0.00	0.16	0.00	0.00	0.00	0.03	0.05	–0.04
function partitioning	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
value profiling	7.83	0.00	0.00	0.00	0.01	0.00	0.00	0.27	0.00	0.00	0.00	0.00	0.05	0.11
superblock formation	7.83	0.00	–0.20	–0.04	0.00	–0.72	0.00	0.71	4.18	–5.73	–0.31	0.11	0.28	0.14
register allocation	0.00	0.22	–0.20	1.07	0.04	0.11	0.00	0.16	1.52	5.62	–0.01	0.31	0.28	0.31
profile driven inliner	0.00	0.00	0.00	0.00	0.79	1.07	8.91	0.00	1.99	0.00	0.00	0.00	0.00	0.35
loop unroll/peel	7.89	19.70	19.11	37.93	0.75	26.45	31.75	7.91	6.06	4.20	16.56	1.51	13.67	8.61
static inliner	0.00	0.00	0.00	0.00	5.13	13.30	12.18	0.00	4.26	0.00	0.00	0.00	0.00	1.41
profile estimation	0.00	0.00	0.00	0.00	–0.98	–0.08	0.00	–0.07	0.00	0.00	–0.32	0.00	0.00	–0.29
profile feedback	15.84	19.38	18.57	37.87	–4.11	26.33	32.64	4.48	4.23	–5.58	13.74	0.51	11.43	6.06
overall	17.07	20.19	18.98	38.06	1.46	26.33	32.53	9.14	11.09	–5.47	16.95	1.67	14.94	9.17