# A New Loop Optimizer for GCC

*Zdeněk Dvořák*
SuSE Labs
`dvorakz@suse.cz, http://atrey.karlin.mff.cuni.cz/~rakdver/`

## Abstract

One of the most important compiler passes is a loop optimization. The GCC's current loop optimizer is outdated and its performance, robustness and extendibility are unsatisfactory. A goal of the project is to replace it with a new better one. In this paper we discuss the design decisions – the choice of used data structures and algorithms, usage and updating of auxiliary information,… Then we describe the current state with emphasis on still unsolved problems and outline the possibilities for further continuation of the project, including replacing the remaining parts of the old optimizer and introducing new low-level (RTL based) and high-level (AST based) optimizations.

## Introduction

It is generally known that most of the time of programs is spent in a small portion of code ([HP]). Those small but critical areas usually consist of loops, therefore it makes sense to expect the optimizations that directly target loops to have a great effect on program performance. Indeed optimizations to improve the efficiency of scheduling, decrease a loop overhead, optimize memory access patterns and exploit a knowledge about a structure of loops in various other ways were devised; see [BGS] for a survey. Certainly no seriously meant compiler may ignore this. GCC contains a loop optimizer that supports the following optimizations:

- Loop invariant motion that moves invariant computations out of loops.
- Strength reduction, induction variable elimination and various other manipulations with induction variables like fitting into machine addressing modes.
- Doloop optimization, i.e. usage of low overhead loop instructions if a target machine provides them.
- Prefetching of arrays used inside loops to reduce cache miss penalties.
- Unrolling of loops to reduce loop overheads, improve the efficiency of scheduling and increase sequentiality of a code.

We refer to this loop optimizer as the old one in the rest of the paper.

The importance of loop optimizations has been recognized for a long time and the old loop optimizer was added to GCC very early (a copyright notice in the `loop.c` file dates it to 1987). The lack of knowledge about the optimization as well as the lack of computing power lead to several design choices that were unfortunate and today cause the optimizer to be much less powerful than it could be. They also cause other problems concerning its robustness, extendibility and restrictions imposed on the other optimizers. This lead us to decide to replace it by a new one by rewriting some parts, adapting some parts for a new infrastructure and extending it by new important optimizations. We refer to the goal of our efforts as the new loop optimizer in the rest of this paper.

The paper is structured as follows: In the section 1 we investigate the structure of the old loop optimizer and problems with it. In the section 2 we discuss goals of the project to replace it and the high-level design choices of the new loop optimizer. Then we continue by providing the detailed description of the current state of the new loop optimizer, including the changes made in the loop analysis. In the following section 3 we describe used data structures and algorithms to update them. In the section 4 we summarize a status of the project, provide some benchmark results and state our future goals.

## 1 The Old Loop Optimizer

The loop optimizer was added to GCC very early. Due to the lack of a computing power (and partially also the lack of knowledge) in those times, it has several features that are quite unusual for modern compilers.

Firstly the loop discovery is based on notes passed from the front-end. This approach is very fast, but the considered loops are therefore required to form a contiguous interval in the insn chain and to fit into one of a few special shapes (of course covering all of the most important cases). The loops created by non-loop constructs (gotos, tail recursion, . . . ) are not detected at all. Optimization passes before the loop optimizer are required to preserve the shape of loops and the placement of loop notes. Most of them fortunately do not modify control flow graph, but those few that do are complicated and restricted by this need.

Additionally sometimes this information is not updated correctly, therefore it must be verified in the loop optimizer itself and the offending loops are ignored. This makes us miss some more optimization opportunities.

The second problem is the handling of jumps inside loops. The global (not specific to a single pass) control flow graph was introduced into GCC very lately (2000), and the loop optimizer works over the insn chain only. Consequently the effects of branches are estimated mostly by simple heuristics and results of loop invariant and induction variable analyses tend to be overly conservative.

As a side issue, the unroller does not update control flow graph, forcing us to rebuild it. This prevents us to gather a profiling feedback before the loop optimizer, as this information is stored in control flow graph. Therefore we cannot use it in the loop optimizer itself and in the previous passes (most notably GCSE and loop header duplication).

The unroller uses its own routines to copy the insn stream, creating an unnecessary code duplication with the other parts of the compiler.

Any single of these problems could probably be addressed separately by modifying the relevant code. Considering them together it seems to be easier to write most of the optimizer again from scratch. Some parts can just be adapted for a new infrastructure (the decision heuristics and execution parts of the invariant motion and induction variable optimizations, the whole doloop optimization pass), but the greatest part has too deeply inbuilt expectations about a loop shape with respect to the insn chain to be usable. We discuss the plans concerning this rewrite in more detail in the following sections.

The source of other complications is the low level of RTL. During the translation to RTL, some of the information about possibility to overflow and types of the registers is lost and we are forced to either rediscover it through nontrivial analysis, use conservative heuristics, produce a suboptimal code containing unnecessary overflow checks or produce a possibly incorrect code. None of these options is particularly good. It would also make dependency analysis quite complicated – it is not present in GCC yet, and the optimizations that require it (the loop reorganization, the loop fusion, ...) are missing. While the current project is mostly RTL based, it will be necessary to address these issues in near future. There are already some efforts for moving the relevant parts of the loop optimizer to the AST level in progress; for more information see section 4.

## 2 Overview of The New Loop Optimizer

There are several basic principles we have decided to follow:

- The passes that form the loop optimizer should be completely independent on each other. They must preserve the common data structures and it should be possible to run them any number of times and in any order (although of course not all orders are equally effective). This approach is completely different from the old loop optimizer one – there the optimizers called each other in non-transparent manner and most of them had assumptions about information gathered by the other ones. While this approach may be slightly more efficient and perhaps simpler at some places with respect to keeping the information up to date during transformations, we prefer our approach due to its cleanness, extendibility and robustness. We have also initially made some parts of the optimizer quite simplistic, and this approach enables us to replace them later by more involved solutions without greater problems.

- We have decided to generally reuse as much of the existing code as possible and eventually extend it for our purposes, rather than creating our own variations of the existing code. Most importantly we used the `cfglayout.c` code for duplicating basic blocks (this should replace two instances of a similar code, one in `unroll.c` and the other one in `jump.c`) and of course the existing `cfgloop.c` code for a loop analysis (after significant changes described below). We are also currently using the code from `simplify-rtx.c` when computing a number of iterations of a loop. In this case we were unfortunately forced to start working on an alternative RTL simplification code for this purpose. The reason is that the goal of `simplify-rtx.c` is in some sense opposite to what we would need. While we need to simplify the RTL expressions into a simple canonical shape, `simplify-rtx.c` code tries to transform it so that it is efficiently computable. Some of the manipulations it does for this purpose (expressing multiplication through shifts) make it unsuitable for our needs, and some conversion we need to do (using distributive law on products of sums) make the resulting code possibly much less efficient than the original one. The two approaches do not seem to fit together very well.

- As much of the information as possible should be kept up to date at any given time. This concerns mostly complicated operations over loops (unrolling, unswitching, ...), where we express them as a composition of simpler operations that preserve the consistent state rather than making them at once and updating the structures afterwards. This makes the code a bit slower, but much easier to understand and debug (many bugs were caught early due to a possibility to check a consistency after every step).

The optimizer itself consists of the initialization, several optimization passes and the finalization. The finalization part is trivial, just releasing the allocated structures. In the following paragraphs we examine the remaining phases in a greater detail.

The initialization and finalization parts are placed in `loop-init.c`. During the initialization, we calculate the following information (that is kept up to date till the finalization):

- A dominator relation is computed. The dominators are used to define and find natural loops and we use them during loop transformations for several purposes, most importantly during the simple loop analysis to determine expressions (conditions) that are executed (tested) in every iteration of the loop. Also we need them to be able to update the loop structure when parts of the code are removed. The decision to keep the dominator relation always up to date turned out to be somewhat disputable. Having them ready at all times is convenient and makes the parts where they are used quite simple, but updating them is relatively non-trivial and quite costly. Most of their usages would be simple to replace without using them at a little extra cost, but their usage during the removal of a code seems to be crucial.

- Natural loops are found. The natural loop is defined as a part of a control flow graph that is dominated by the loop's header block and backreachable from one of the edges entering the header, called the latch

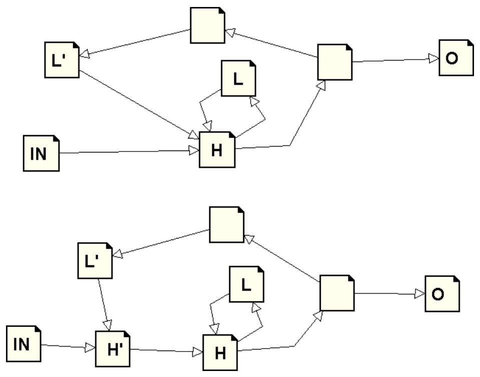

Figure 1: Creating nested loops from loops with shared header.

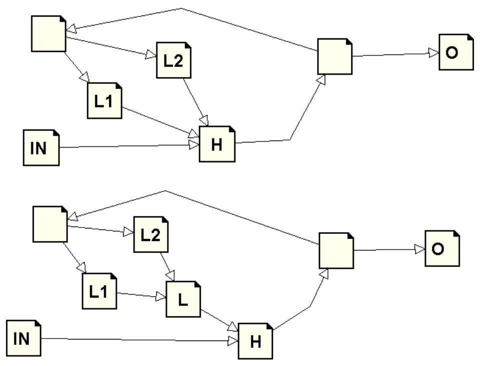

Figure 2: Merging loops with a shared header.

edge. Note that this definition makes it possible for several loops to share the same header block. We do not want to have to handle them specially, so we split the loop header in this case. There are two ways to split the header (figures 1 and 2) – one of them merges the loops together, while the other one creates the nested loops. It is impossible to recognize which of these cases matches the reality just from a control flow graph, and even looking at the source code does not help too much (this kind of loops is often created by continue statements, and it is hard to recognize what behavior describes this situation better). If we have a profile feedback available, we use it to determine whether one of the latch edges is much more frequent than the other ones, i.e. if it behaves like an inner loop, and create the inner loop in this case (this is sometimes called the commando loop optimization). Otherwise we just merge the loops.

- `cfg_layout_initialize` is called to bring the instruction chain into a shape that is more suitable for the transformations. This function removes the unconditional jumps from the instruction stream (the information about them is already included in the control flow graph) and makes it possible to reorganize and manipulate basic blocks in much easier manner.

- Loops are canonicalized so that they have simple preheaders and latches. By this we mean that:

  – Every loop has just a single entry edge and the source of this entry edge has exactly one successor.

  – The source of latch edge has exactly one successor.

  This makes moving a code out of the loop easier, as there is exactly one place where it must be put to (the preheader) and we can put it there without a fear that it would be executed if we do not enter the loop. It also removes the singular case of a loop that consists of just one block. A quite important fact is that the loop latch must now belong directly to the loop (i.e. it cannot belong to any subloop) and the preheader belongs directly to the immediate superloop of the loop (it could belong to a sibling loop if it had more than one successor).

- The irreducible regions are marked. A region of a control flow graph is considered irreducible if it is strongly connected and has more than one entry block (i.e. it contains a depth first search back edge, but the destination block of this edge does not dominate its source, so the region fails to be a natural loop). The irreducible regions are quite infrequent (it is impossible to create them in structured languages without use of a goto statement or a help of the compiler), but we must be able to handle them somehow. In the new loop optimizer they are mostly ignored, just taking them into account during various analyses. The information about them is quite easy to keep up to date unless we affect their structure significantly. This may occur in very rare cases during the unswitching or the complete unrolling. In some of these cases we have resigned on updating the information and rather recompute them from scratch – it is quite fast (just a depth first search over a control flow graph) and much less error prone than to attempt to handle the case that we would not be able to test properly (it is almost impossible to construct a suitable testcase).

The optimization passes are placed in separate files. The currently available optimization passes are:

- Loop unswitching (in `loop-unswitch.c`) – if there is a condition inside a loop that is invariant, we may create a duplicate of the loop, put a copy of the condition in front of the loop and its duplicate that chooses the appropriate loop and optimize the loop bodies using the knowledge of a result of this condition. There are a few points worth the attention. The first is a code growth – if there is a loop with $k$ unswitchable conditions, we end up with $2^k$ duplicates of the loop. This is not really a problem in practice – the opportunities for unswitching are rare. Also in most of the cases when we have more than one unswitchable condition per loop the values tested in them are identical and they are therefore eliminated already during the first unswitching. (Just for sure the number of unswitchings per loop is limited). The other is testing for invariantness of the condition. As the new loop optimizer is placed after GCSE (and also the old loop optimizer's invariant motion), it is sufficient to just test that the arguments of the condition are not modified anywhere inside the loop.

- Loop unrolling and loop peeling (placed in `loop-unroll.c`). While it would correspond more to our philosophy to have this pass split into several ones, the code and computation sharing between them is so large that it would be impractical. Anyway they are still completely independent and they could be split with a little effort. We perform the following optimizations:

  – Elimination of loops that do not roll at all – this is somewhat exceptional, as this does not increase code size (in fact it decreases it). For this reason we perform this transformation even for non-innermost loops, unlike the other ones.

  – Complete unrolling of loops that iterate a small constant number of times (a loop is eliminated in this case too, but at the cost of a code size growth).

  – Unrolling loops with a constant number of iterations—we may peel a few iterations of the loop and thus ensure that the loop may only exit in a specified copy, therefore enabling us to remove now useless exit tests. For most of the loops we leave the exit in the last copy of the loop body—the exit is usually placed at the end of loop body, and all copies may be merged into a single block in this case. In the rare cases when this is not true we leave the exit in the first copy—in this case it is a bit easier to handle loops of a form `for (i=a; i < a+100; i++)`, where the number of iterations may be either $100$ or $0$ (in the case of an overflow).

  – Unrolling loops for that the number of iterations may be determined in runtime – the situation is similar here, except that the number of iterations to perform before entering the unrolled loop body must be determined in runtime. The number of iterations to be performed is chosen through a switch statement-like code.

  According to some sources ([DJ]), in both of these cases it is preferable to place the extra iterations after the loop instead due to a better alignment of data (this might also be important if we were doing autovectorisation). This can only be done if the loop has just a single exit and modifications of the loop are more complicated. Also handling of overflows and other degenerate cases becomes much harder. It could however be done for constant time iterating loops with a little effort.

– Unrolling of all remaining loops – this transformation is a bit controversial. The gains tend not to be large (scheduling may be improved and rarely some computations from two consecutive iterations may be combined together), and sometimes we even lose efficiency (due to negative effects of a code growth to instruction caches and an increased number of branches to branch prediction). We only do this if specifically asked to, and even then only if the loop consists of just a single basic block.

– Loop peeling – the situation is similar (additionally we hope that the information about initial values of registers can be used to optimize the few first iterations specially). We gain most for loops that do not iterate too much (optimally we should not even enter the loop). To verify this, we use a profile feedback and therefore perform this transformation only if it is present.

As was already mentioned, we perform these transforms on innermost loops only. This is not a principal restriction (the passes are written so that they handle subloops), but the ratio of a code size growth to a performance gain is bad then, and also duplicated subloops would be more difficult for branch prediction in processors.

The old loop unroller also performs the induction variable splitting to remove long dependency chains created by unrolling that negatively impact scheduling and other optimization passes. We instead leave this work to the webizer pass that is much more general.

There are three basic problems to solve. Firstly there is the code growth. All of the unrolling-type transformations naturally increase a code size. While the greater number of unrollings generally increases effect of the optimization, it also increases a pressure on code caches. It is therefore important to limit the code growth. There are adjustable thresholds that limit the size of resulting loops as well as the maximal number of unrollings. We also use a profile feedback to optimize only relevant parts and try to limit transformations for that gains are questionable in cases when we believe that they might spoil the code instead (for example the loop peeling is not performed without a profile feedback that would suggest that the loop does not roll too much).

A more appropriate solution might be a loop rerolling pass run after scheduling that would revert the effects of a loop unrolling in case we were not able to get any benefits from it.

Secondly we need the analysis to determine a number of the loop's iterations. Currently we use a simplistic analysis that for each exit from the loop that dominates the latch (i.e. is executed in every iteration) checks whether the exit condition is suitable – i.e. if it is comparison where one of the operands is invariant inside the loop and the other one is set at exactly one instruction that is executed exactly once per loop iteration. For such condition we then check whether the variable is increased by constant and attempt to find its initial variable in an extended preheader of the loop (i.e. basic blocks that necessarily had to be executed before entering the loop). Using the simplification machinery from `simplify-rtx.c` we then determine the number of iterations. This turns out to be sufficient in most cases, but things like multiple increases of the induction variable prevents us from detecting the variable. Also often the initial value of the variable is assigned to it earlier, preventing us from recognizing the loop as iterating a constant number of times. Induction variables that iterate in a mode that is narrower than their natural mode are not handled, which causes problems on some of the 64 bit architectures where int type is represented this way. We are currently working on the full induction variables analysis that solves all of these problems.

Thirdly we must decide how much we want to unroll the loop. Currently we take into account just a code growth, thus we unroll the bigger loops less times. For constant times iterating loops we also attempt to adjust the number of unrollings so that the total size of the code is minimal. In other cases we use the heuristic that says that it is good to unroll number of times that is a power of two (because of better alignments and other factors). See the section 4 for discussion of the possible extensions of this scheme and the estimation of gains obtainable by using some better methods.

- Doloop optimization – this pass is just an adaptation of the old loop optimizer's doloop pass that was written by Michael Hayes. The structure of the pass was quite clear and there were no major problems with the transfer. This adaptation of the pass is still only present on rtlopt-branch, due to a bad interaction with the new loop optimizer. This is caused by a overly simplistic induction variable analysis used and should be solved by the improved induction variable analysis that is currently being written.

## 3 The Data Structures

In this section, we discuss the structure to represent the loops as well as other auxiliary data structures used in the new loop optimizer. We also describe the algorithms used to update them.

We consider a loop $A$ a *subloop* of a loop $B$ if a set of basic blocks inside the loop $A$ is a strict subset of a set of basic blocks inside the loop $B$. Because we have eliminated the loops with shared headers, the Hasse diagram of a partial ordering of loops by the subset relation is a forest. To make some of the algorithms more consistent, we add an artificial root loop consisting of the whole function body (with an entry block as a header and an exit block as a latch). We maintain this loop tree explicitly. For each of the nodes of the tree we remember the corresponding loop's header and latch. But we do not remember the set of basic blocks that belong to it – if we need to enumerate the whole loop body, we use a simple backward depth first search from its latch, stopping at its header.

To be able to test for the membership of a basic block to the loop we maintain the information about the innermost loop that each basic block belongs to. To speed up the testing for a not necessarily immediate membership to a loop (i.e. including membership to any subloop of the loop), we also maintain the depth in the loop tree and an array of parents for each node in the loop tree. Maintaining the arrays of parents enables us to respond to these queries in a constant time, but makes speed of all updates proportional to the depth of the tree. This works well in the practice, as the tree is usually quite shallow and the structure of a loop tree does not change very often.

Updating the loop tree is straightforward during the control flow graph transformations we use. Most of the optimizations do not change the structure at all. The exception is the loop unrolling and peeling type transformations if some subloops are duplicated (it cannot really occur just now because we optimize only innermost loops, but the code can handle this situation for case we changed this decision) or the unrolled loop is removed, but all of these cases are easy to handle. Note that some of them may create new loops if irreducible regions are present. We ignore these newly created loops (still having them marked as irreducible) – this is conservatively correct and this situation is so rare that it does not deserve any other special handling.

As described in the previous section, there are two further pieces of information we keep up-to-date – the dominators of basic blocks and the information about irreducible regions.

We represent the dominators as the in-branching of immediate dominators. We represent this in-branching using ET-trees. This structure was chosen due to its flexibility – it enables us to perform all relevant operations asymptotically fast (updates and queries for dominance in logarithmic time, finding the nearest common dominator of a set of blocks and enumerating all blocks that are immediately dominated by a given block in a time proportional to the size of the relevant set times a polylogarithmic time). The multiplicative constants however turned out to be quite high and we are considering replacing the structure by some perhaps less theoretically nice but more practical (e.g. a depth first search numbering with holes).

Updating of the dominators in general is not easy. During the transformations we perform we are usually able to handle it by using the fact that they are of a special kind (respecting the loop structure that itself reflects the structure of dominators). In a small portion of cases when we are not able to do it (or the rules to determine how the dominators change would be too complicated) we use a simple iterative approach (similar to [PM]) to update the dominators in the (usually) small set of basic blocks where they could be affected. As already mentioned before, we also consider not keeping the dominators at all and solving the cases when they are currently used without them.

The irreducible regions are determined as strongly connected components of a slightly altered control flow graph. For each loop we create a fake node. Entry edges of the loops are redirected to these nodes, exit edges are redirected to lead from them – this ensures that the parts of the irreducible regions that pass through some subloop are taken into account only in the outer loop. We remember this information through flag placed on the edges that are a part of those strongly connected components. This is sufficient to update the information effectively during the most of the control flow graph transformations. The only difficult case is when a loop that is a part of an irreducible area is removed. We would have to propagate the information about irreducibility through the remnant of its body then. While it could be done, it would be quite difficult to handle all problems (subloops, other irreducible regions). Instead, we simply remark all irreducible regions using the algorithm described above (this situation is quite rare and the algorithm is sufficiently fast anyway).

| Benchmarks | Base Ref Time | Base Run Time | Estimated Base Ratio | Peak Ref Time | Peak Run Time | Estimated Peak Ratio |
|---|---|---|---|---|---|---|
| 164.gzip | 1400 | 306 | 458 | 1400 | 291 | 480* |
| 175.vpr | 1400 | 452 | 310 | 1400 | 452 | 310* |
| 176.gcc | 1100 | 306 | 360 | 1100 | 299 | 368* |
| 181.mcf | 1800 | 821 | 219 | 1800 | 815 | 221* |
| 186.crafty | 1000 | 174 | 574 | 1000 | 174 | 575* |
| 197.parser | 1800 | 534 | 337 | 1800 | 534 | 337* |
| 252.eon | 1300 | 201 | 648 | 1300 | 199 | 652* |
| 253.perlbmk | 1800 | 338 | 533 | 1800 | 335 | 538* |
| 254.gap | 1100 | 280 | 393 | 1100 | 277 | 398* |
| 255.vortex | 1900 | 414 | 459 | 1900 | 410 | 464* |
| 256.bzip2 | 1500 | 431 | 348 | 1500 | 428 | 351* |
| 300.twolf | 3000 | 902 | 333 | 3000 | 878 | 342* |
| Est. SPECint_base2000 | | | 398 | | | |
| Est. SPECint2000 | | | | | | 403 |

Base flags: -O2 -march=athlon -malign-double -fold-unroll-loops
Peak flags: -O2 -march=athlon -malign-double -funroll-loops

Figure 3: SPECint2000 results for rtlopt-branch on Athlon, 1.7 GHz

| Benchmarks | Base Ref Time | Base Run Time | Estimated Base Ratio | Peak Ref Time | Peak Run Time | Estimated Peak Ratio |
|---|---|---|---|---|---|---|
| 164.gzip | 1400 | 621 | 225 | 1400 | 605 | 232 |
| 175.vpr | 1400 | 857 | 163 | 1400 | 854 | 164 |
| 176.gcc | 1100 | 624 | 176 | 1100 | 618 | 178 |
| 181.mcf | 1800 | 1354 | 133 | 1800 | 1361 | 132 |
| 186.crafty | 1000 | 285 | 350 | 1000 | 275 | 364 |
| 197.parser | 1800 | 930 | 194 | 1800 | 932 | 193 |
| 252.eon | 1300 | 321 | 405 | 1300 | 331 | 393 |
| 253.perlbmk | 1800 | 538 | 335 | 1800 | 556 | 324 |
| 254.gap | 1100 | 426 | 258 | 1100 | 420 | 262 |
| 255.vortex | 1900 | 817 | 233 | 1900 | 810 | 235 |
| 256.bzip2 | 1500 | 770 | 195 | 1500 | 774 | 194 |
| 300.twolf | 3000 | 1709 | 176 | 3000 | 1699 | 177 |
| Est. SPECint_base2000 | | | 224 | | | |
| Est. SPECint2000 | | | | | | 225 |

Base flags: -O2 -march=athlon -fold-unroll-loops
Peak flags: -O2 -march=athlon -funroll-loops

Figure 4: SPECint2000 results for mainline on Duron, 800 MHz

## 4 The Current State And Further Plans

Everything described above in the paper (except for the doloop optimizer adaptation) is already merged in the GCC mainline and will be in GCC 3.4. The new loop unroller in connection with webizer and other improvements present on rtlopt-branch outperforms the old one on i686 and even without the webizer the results are comparable (see figures 3 and 4 for results on SPECint2000 testsuite). Its simple procedure to count the number of iterations beats the old loop optimizer's one (it detects 52 loops as iterating a constant number of times on the gap benchmark compilation as opposed to 39 loops the old loop optimizer did). The total number of loops detected is a bit surprisingly almost the same – 3292 by the old loop optimizer, 3298 by the new one – writers of GCC have apparently done very good job in keeping the front-end information about the loops accurate.

We are still quite far from our final goal – fully replacing and removing the old loop optimizer. What remains is to replace or adapt induction variable optimizations (the invariant motion can be solved by GCSE instead) and to solve the problems described below.

While the results from i686 look quite promising, the new loop optimizer has problems on the other architectures. Some of the 64-bit architectures must represent 32-bit integers as subregs of 64-bit registers. The simplistic analysis to determine a number of iterations of the loops is not yet able to handle this case, so the unroller is useless here. This should be solved by introducing the new induction analysis that is needed to replace the induction variable optimization parts anyway.

On some other architectures quite important performance regressions were reported. They might be partially caused by absence of the webizer pass in mainline. We are currently investigating other reasons.

The interesting problem with the new loop unroller is determining whether and how much we should unroll or peel a given loop. There are several possible criterion:

- To decide whether to optimize at all, we use a profile feedback. Not optimizing in cold areas reduces the code growth a lot. To decide whether to peel or to unroll, we try to estimate the number of iterations of a loop using the feedback and to peel a sufficient number of iterations from a loop so that the loop is not entered at all most of the times. We also measure histograms of first few iterations of the loops and use it to determine this more precisely on rtlopt-branch, but the effects are not significant.

- The effects on instruction cache seem to be quite important. There are some works describing how to take them into account ([HBK]), but they would require a global program analysis and it seems questionable whether they would be useful at all. For now we cannot do anything but to attempt to limit the code size growth.

  Similarly duplication of loops whose bodies contain many branches may also affect the performance negatively, as the created jumps increase the pressure on the CPU's branch prediction mechanisms. Sometimes these jumps may also may behave less predictably than the original ones.

- From a scheduling point of view, it would make sense to prefer unrolling loops that contain instructions with long latencies. It might also be useful to take a register allocation into account, attempting to minimize the number of registers needed for computing simple recurrences.

Currently we use only a very simple heuristics to take some of the effects mentioned above into account. To estimate the possible gains of using better methods, we wrote a code that attempts to determine the best possible number of unrollings for each of the loops. It adds a code for each of the loops that measures the total time spent inside it. Then for $i$ between $1$ and some upper bound, we unroll all loops $i$ times and gather the profiling data. Finally we choose the best of these times for every loop as the right number of iterations to unroll. This is far from optimal (the added profiling code changes the performance characteristics of a compiled program a lot and the optimal numbers are also dependent on how other loops are unrolled, so measuring them when all are unrolled the same number of times is not completely right), still we achieved about 2% speedup on SPEC2000 this way on i686.

Adapting the rest of old loop optimizer seems to be quite straightforward now. New induction variable analysis pass is just being tested on rtlopt-branch, the next step is either to use it to produce induction variable descriptions suitable for the old induction variable optimization pass, or (more likely) to write a new one, heavily reusing the parts of the old one.

There are additional loop optimizations that should be added to GCC, including

- loop reorganization that makes accesses to arrays more sequential by swapping an order of nested loops if possible.
- loop fusion that joins adjacent loops that iterate the same number of times (perhaps after a small adjustment), to reduce an overhead of loop creating instructions.
- loop splitting that inversely splits the loops into several smaller ones if we know that we are able to optimize them better this way.
- autovectorisation, i.e. usage of SIMD instructions on arrays processed in loops.

All of those (and several other less important) optimizations require a dependency analysis to determine whether it is indeed possible to reorganize computations as needed. It would be pretty painful to determine this on the RTL level, as information about types of variables is almost lost here (partially recoverable only through a complicated analysis) and so is some of the information about overflows. Also the loop reorganization needed would be quite complicated on the RTL level. This makes them more suitable for the AST level. We hope to be able to start a work on them in a few months.

Other optimizations should be better done on AST level from similar reasons, including a part of induction variable optimizations that does not use a machine specific information (like a knowledge of addressing modes etc.) and possibly unrolling and unswitching. There are already some efforts for moving the relevant parts of the loop optimizer to the AST level in progress (Pop Sébastian have recently altered the loop recognition code to work both on RTL and AST levels).

## Acknowledgments

The project is based on the GCC code written by hundreds of volunteers. The most of contributions to the loop analysis code we have built upon were by Daniel Berlin, Michael Hayes and Michael Matz. Part of the new loop optimizer was written during the "Infrastruktura pro profilem řízené optimalizace v GCC" ("Infrastructure for profile driven optimizations in GCC") software project at Charles University, Prague ([DHNZ], [DHNZ-doc]) together with Josef Zlomek, Pavel Nejedlý and Jan Hubička under leadership of David Bednárek and later continued with support of Suse Labs. I would also like to thank Richard Henderson for providing a useful feedback during the merging of the new loop optimizer to the mainline.

## References

[BGS] David F. Bacon, Susan L. Graham and Oliver J. Sharp, *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys 26 (1994) p. 345–420.

[DJ] Jack W. Davidson and Sanjay Jinturkar, *An Aggressive Approach to Loop Unrolling*, Technical Report CS-95-26, Department of Computer Science, University of Virginia, Charlottesville, June 1995.

[DHNZ] *The "Infrastruktura pro profilem řízené optimalizace v GCC" project specification*, `http://ksvi.mff.cuni.cz/~holan/SWP/zadani/gccopt.txt`

[DHNZ-doc] *The "Infrastruktura pro profilem řízené optimalizace v GCC" project documentation*, `http://atrey.karlin.mff.cuni.cz/~rakdver/projekt/`

[HBK] K. Heydemann, F. Bodin, P. Knijnenburg, *Global Trade-off between Code Size and Performance for Loop Unrolling on VLIW Architectures*, Publication Interne 1390, IRISA, Institut de Recherche en Informatique et Syst‘emes Al'eatoires, March 2001.

[HP] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc, San Matea, CA, 1990.

[PM] Paul W. Purdom, Jr. , Edward F. Moore, *Immediate predominators in a directed graph*, Communications of the ACM, v.15 n.8, p.777-778, Aug. 1972