Infrastructure for Profile Driven Optimizations in GCC Compiler

Zdeněk Dvořák, Jan Hubička, Pavel Nejedlý, Josef Zlomek

April 12, 2002

Contents

Part I

Introduction

Chapter 1

Goals and Motivation

GNU Compiler Collection (GCC) is major free-software (or open-source) compiler. It is an aged and large project that gained lots of popularity because of simple retargetability and easy extensibility at front-end side. Since the early 80ties when the project officially started till today it has been targeted to over 90 CPU architectures[1] and many more operating systems, CPU flavors (the 32bit and 64bit versions of popular architectures are counted as a single target) and assembler/object file formats. Today GCC is commonly the compiler of choice for embedded targets and is used as major compiler in number of popular operating systems.

On the front end side, in addition to C currently mostly compliant with C99 standard, GCC has been extended to support C++ (GCC 3.0 is one of first compilers supporting ISO C++ standard and IA-64 exception handling specification), ADA, Fortran77 (Fortran9x front-end is being worked on), Chill and in separate tree Pascal (both ISO Pascal and Borland Pascal-like dialects).

We found GCC development a challenging task and since the sources are publicly available, we decided to participate on it. While choosing the goals of GCC related software project, we had to take into account a number of factors:

The major problem of GCC for years was little extensibility in terms of new optimization passes. The classical optimization passes — jump optimization, common subexpression elimination, instruction combiner, loop optimizer and priority based register allocator have evolved over the years into very powerful optimizers. For instance the common subexpression elimination pass now does de-facto global value numbering, or instruction combiner utilizes powerful algebraic simplifier of expressions. Also the design has aged and further extensions have become difficult.

Adding new optimization passes was not very successful. In fact only a few new optimization passes were added in the last decade. One of the more successful was the addition of scheduler pass in 1992, but it took long time until it was adopted by machine descriptions.

In 1997 GCC project went through development model restructuralization and became much more open, having public CVS[2] tree, weekly snapshots and

[1]counting only those present in the official tree, number of other ports were obsoleted and dropped or never integrated making the overall count even higher.

[2]Concurrent Version System, the package to maintain multiple development trees of same project in a single repository.

group of maintainers reviewing and applying patches. This new scheme allowed more rapid development and some new optimization passes were integrated. Considerable fraction of these may be considered unsuccessful.

A nice example of these unsuccessful integrated optimization passes is the global common subexpression elimination pass (GCSE) dated to 1997. It produced code that required more registers and ran slower than without the new pass, that was attributed to busy code motion algorithm used to implement partial redundancy elimination. Lazy code motion code came in 1998, but GCSE still was not powerful enough to change performance of common benchmarks in a positive way. The problem were limitations of register allocation and limited alias analysis support in compiler.

After rewrite of the i386 back-end used by all x86 platforms like i386, Pentiums or Athlons in 1999 the GCSE became mostly disabled for this platform as it did not understand more detailed intermediate representation of the back-end. In 2000 the work started on Single Static Assignment (SSA)[3] support in GCC that was expected to make GCSE (and other optimizations) implementation easier, but this effort was mostly dropped after failure to use SSA to implement a simple dead code removal pass. Intermediate language used by GCC at the present, Register Transfer Language (RTL), contains some constructs making SSA representation of program impossible[4].

In 2001 GCSE was partially modified to handle new representation of i386 instructions and a support for load/store motion was integrated. Even today in the official tree, GCSE is not able to do partial redundancy elimination on i386 instructions modifying flag registers (majority of them) and store motion implementation is disabled due to design problems, so the old pseudo-global CSE and GCSE are both run resulting in large compilation time overhead.

This can not be attributed to the wrong design of GCSE, that by itself is nice and clean piece of code, but to the fact that GCC contains almost no support for the optimization passes and the RTL language used is too low level which makes the implementation of optimizers unacceptably difficult. (Author believes that RTL is a good intermediate language that fits perfectly for the purposes of late passes such as instruction combining, register allocation or instruction scheduling it was designed for).

In our project we decided not to concentrate on adding new optimization features, as this looked like a lost battle, but on modernizing the compiler itself. We believe that if things are done properly, lots of effort on implementing new optimizers can be saved. We had to find our room in other outgoing projects of that time:

Loop optimizer rewrite, Michael Hayes

Michael was rewriting the current loop optimizer, using information provided by front-end about loop constructs and had problems because earlier passes mangled the information in a way that loop optimizer often had to skip the loop.

His new loop implementation uses natural loop discovery code on control flow graph to find the loop constructs. He has contributed the natural loop

[3]The modern intermediate representation where each variable has exactly one definition in the program making many analyses easier.

[4]RTL requires same register to be both used and defined in the single instruction in some constructs.

discovery code and new DU/UD analysis module, but has not finished the loop optimizer rewrite itself yet.

Mid-level RTL, Jeff Law
Register Transfer Language, the intermediate program representation used by GCC, is extremely low level. It allows to describe lots of details about target platforms and is the source of GCC's easy retargetability, however it is very complicated for optimizers to handle it and makes it impossible to convert program into SSA that is used by many modern algorithms. Jeff Law worked on higher level RTL, that looked syntactically identical to the low level RTL but hid all unnecessary target-specific features.

Abstract Syntax Tree optimizers, Diego Novillo
Recent rewrite of C++ front-end made it possible to keep representation of the whole function in abstract syntax tree format, instead of lowering the function statement by statement to RTL representation. This made it possible to write optimizations at tree level making implementation of high level optimizers much easier. Mark Mitchell has donated new implementation of function inlining code based at trees and Red Hat and CodeSourcery developers have been working on modifying C front-end to use the same representation.

Diego has started separate development branch targeting to implement an infrastructure for high level optimizers — control flow graph over tree representation. He plans to implement single static assignment form, alias analysis, dependency analysis, high level loop optimizers and more.

DFA scheduler, Vladimir Marakov
Vladimir has implemented new automaton based hazard recognizer for scheduler allowing much closer description of target architecture. On the newly created branch some machine descriptions have been converted to use this new format. Now he plans to implement proper modern global scheduling pass.

New register allocator branch, Daniel Berlin and Michael Matz
Patents pending on the Chaitin's graph coloring idea and most of other modern register allocation algorithms forced GCC to stay with priority based register allocator from late 80ties. Daniel and Michael have implemented new, iterated register coalescing based algorithm. The patent part of this is currently sorted out.

Several front-end and retargeting projects
Just for sake of completeness we should mention that several projects were active in the front-end area and several ports were in the development. These areas basically did not influence our work, except for Fortran9x project that introduced notion of multiple entry points to single function.

Concerning the control flow graph (CFG), one of basic preexisting pieces of code needed for our project, each pass had its own implementation of CFG until 1998. Some passes, such as reg-stack[5] even did its own liveness analysis and nontrivial CFG transformations, such as critical edge splitting.

[5]Pass converting RTL into form expected by i387 stack-like registers.

In 1998 Richard Henderson provided a new, more flexible CFG implementation for data-flow module and revamped register allocation and related passes to it. Several other passes, scheduler and reg-stack were converted later, but still the majority of the passes did not use it and even worse, invalidated it by modifying the instruction stream without updating the CFG datastructure. Thus these structures had to be rebuild before every use.

We decided to concentrate on pushing the control flow graph (CFG) data structure further into GCC implementation and attempt to increase sharing of infrastructure over GCC passes. Just for illustration, before starting project, GCC contained 11 different implementations of function to remove instruction from the chain and many places in compiler did the work manually.

Since only cleaning up half a million lines of code is a dreadful task, we have decided give our work a limited, but important and hopefully fruitful goal — pushing profile information (see chapter 14.3) into GCC optimization passes. Impact, DEC a Intel compiler teams have reported that profile based optimizations bring substantial speedups (see for instance [2]) and profile information is best stored directly in the CFG itself, making the effort of converting existing passes to CFG more appealing.

Thus we decided to extend the lifetime of the control flow graph over the majority of optimization passes and update existing code to profile applications to allow reading data back into the compiler. Doing so required to reimplement some optimization passes, clean up others and most importantly develop an infrastructure for other GCC developers to easily manipulate with CFG representation and underlying RTL instruction chain.

To derive some more fruits from our work, we decided to develop some new optimization passes based on new CFG and profile code as proof of the concept. We also decided to redesign and improve current static profile estimation pass and the profiler itself as described later in this document.

Previously, a team of compiler developers at Intel extended GCC to better support their i960 Intel chip. Their extensions include some of profile based optimizations, trace scheduling and register allocation to name the most important. To minimize amount of changes their design decision was to store the profile information inside instruction stream itself instead in the flow graph structure that is in sharp contrast with our implementation. Intel's implementation was never integrated into official GCC tree.

Our effort was to implement similar optimizations (and more) but in the form acceptable for GCC maintainers. We created a branch called cfg-branch on the official CVS server holding GCC sources and all the patches installed had been sent to official mailing list, so the GCC maintainers had chance to review the code briefly and point out important problems, when present.

In our development protocol, all important patches were first reviewed by other developer on the internal mailing and once ready they were sent to the official mailing list and installed to the cfg-branch tree. The comments from other GCC developers and maintainers sometimes requested changes that were integrated and once code had proven to be useful and working, it was sent for integration to mainline tree. This scheme is similar to primary and secondary review model used by major GCC companies. Bugfixes were allowed to go into tree without approval.

Due to limited manpower of GCC maintainers, the infrastructural changes had priority over new optimization passes, to allow other developers to use our work. Additionally our tree was synchronized with mainline tree on weekly basis allowing us to more easily prepare and test patches for GCC mainline and avoiding risk of conflict with other development project integrated to the mainline.

We also tested our changes by bootstrapping the compiler[6], running the GCC testsuite and daily benchmarking using SPECint2000 with more comprehensive testing run weekly.

[6]Bootstrapping the compiler is a process, where the compiler is used to compile itself.

Chapter 2

Installation

To install GCC to your POSIX-like system you should follow instructions described in GCC installation manual[1]. For GNU/Linux, quick installation instructions for the GCC source tarballs present on CD follows:

To successfully compile GCC, you need recent version of bison, autoconf, working C compiler, make and shell sh. All these packages are present in major Linux distributions today, or available at GNU homepage[1].

For outdated systems you will need to install new binutils first. You can install binunits this way:

```
tar xzvf <path>/binutils.tar.gz
mkdir build
cd build
../binutils-1.12/configure --prefix=<dir>
make
make install
```

To install GCC itself, you need:

```
tar xzvf <file_name>
mkdir build
cd build
../gcc/configure --disable-checking --prefix=<dir>
make bootstrap
make install
```

To debug programs generated by new GCC you need recent GDB that is not present in most Linux distributions yet. To install you do:

```
tar xzvf <path>/gdb.tar.gz
mkdir build
cd build
../gdb-5.1.1/configure --prefix=<dir>
make
make install
```

where ¡dir¿ is a directory where you want GCC to be installed.

[1] http://www.gnu.org

Chapter 3

User Visible Changes

We did minimal changes to GCC from user point of view as we use existing command line options for profiling and feedback based compilation. To profile program, compile program with option `-fprofile-arcs` (for the current GCC snapshots you may need `--static` as well[1]) and run it on the train inputs. Once program is profiled, optimized binary can be produced by compilation with option `-fbranch-probabilities`. Other compiler options must be the same in both passes. Don't forged to remove the profile files (`*.da`) afterwards to avoid them being merged to future profiles of the program.

Unlike previous versions of GCC, `-fbranch-probabilities` has positive effect on generated code[2]. We tested our work on i386/GNU/Linux platform, but the majority of platforms should be functional in 3.1 and 3.2 development trees. In cfg-branch tree we briefly tested Power-PC and Sparc and found it functional.

On CFG-branch we added following command line options to control new optimization passes:

`-fweb` Enables webizer pass. Enabled by default at `-O2` level of optimization.

> Webizer pass improves register allocation and common subexpression elimination in cases where single variable is used in multiple contexts, like `i` as counter in multiple loops.

`-ftracer` Enables tracer pass. Enabled by default at `-O2` level of optimization if profile feedback is present.

> Tracer performs code duplication in order to help other optimizers. The resulting code is larger, but should run faster unless code cache limits are hit.

`-freorder-blocks` Enables software trace cache pass. Enabled by default at `-O2`. This option existed in GCC since version 3.0.0 and enabled old branch reordering pass.

[1] This is caused by the conflict in shared GCC runtime library, libgcc, and will go away once GCC is released and version updated.

[2] Older versions of GCC read data back in mangled form and attempted to use them for code reordering which resulted in poor produced code or crashed during the progress.

Basic block reordering reduces amount of taken conditional jumps in code resulting in better instruction decoder performance and smaller code cache footprint.

-freorder-functions Enables software trace cache pass. Enabled by default at -O2.

Function reordering further improves code locality and avoids code cache conflicts. This option has effect only when profile feedback is available and target assembler supports named sections.

-fnew-unroll-loops Enables unrolling of simple loops. Enabled by default at -O3 level of optimization.

Function unrolling duplicates loop body several times to improve other optimizations and instruction decoder performance. By default unrolling is done only for loop where iteration counter can be identified. **--param max-unrolled-insns=***n* and **--param max-unroll-times=***n* may be used to control amount of unrolling. First option specifies the number of instructions in the loop body unroller is attempting to reach, while the second limits the number of copies of the loop body done.

-fnew-unroll-all-loops Enables unrolling of all loops. Enabled by default at -O3 level of optimization.

Same as **-fnew-unroll-loops**, but all loops are unrolled.

-fpeel-loops Enables loop peeling. Enabled by default at -O3 level of optimization.

Function peeling duplicates a loop body in the front of loop itself. For loops with small average iterations counts it can effectively avoid the loop. Peeled loop body can also be better optimized by other optimization passes and scheduled into the code just before loop.

Again **--param max-peeled-insns=***n* and **--param max-peel-times=***n* options can be used with analogous meaning to the **-fnew-unroll-loops** parameters.

-funswitch-loops Enables loop unswitching. Enabled by default at -O3 level of optimization.

Loop unswitching avoids invariant conditionals in the body of loop by duplicating the loop body and moving the conditional into the header. This usually results in better performance and larger code size.

-fmidlevel-rtl Enables midlevel RTL. Enabled by default for i386 architecture, disabled otherwise.

Midlevel RTL is an alternate intermediate code representation in GCC that may be used for more aggressive optimizations. For some targets, midlevel RTL is required for loop unswitching and loop unrolling to be effective.

-funsafe-profile-arcs Disables thread safe profiling.

`-fvar-tracking` Enables accurate debug output to be generated. No released version of GDB is able to read the data yet, so it is desirable to disable it when you use GDB 5.2.x or older.

This option is on by default.

Effect of all options can be negated using `-fno-` prefix.

We have added an attribute `noprofile` for disabling profiling (see [1] for details).

Chapter 4

GCC Design Overview and Our Modifications

In this chapter we provide a short overview of design of GCC and our changes.

4.1 Intermediate Languages

GCC uses two different intermediate languages to represent the compiled functions. In early compilation stages abstract syntax trees (in future referred simply as "trees") are built as the result of parsing source code. Once whole function body is parsed, or in an old-fashioned front-end at statement basis[1], it is lowered to the very low level register transfer language (RTL) representation, where currently the majority of optimizations is done.

4.1.1 Trees

The trees represent program behavior very closely to the source form, for instance C ternary operator `?` do have different representation from `if` construct or `for` loop different from `while`. Trees are also partly front-end specific (each front-end extends the original C implementation by new nodes) making writing optimizers on this representation difficult.

Earlier mentioned AST–branch has been created to update front-ends to lower representation to common SIMPLE tree form and perform optimizations. We hope this effort will eventually result in high level optimizers for GCC — inter-procedural optimizer, memory hierarchy loop transformation etc., but currently the code is in preliminary design stages and our decision was not to interfere our efforts with AST–branch.

4.1.2 Register Transfer Language

RTL is the low level language used to represent program in later stages of compilation. It has form of gradual lowering implemented due to instruction

[1]This very bad design decision came from attempt to make impossible adding commercial front-ends to the free software GCC back-end and is source for number of problems. An ongoing effort is made to update all front-ends to avoid this scheme.

splitting, so in the final stage, there is usually one to one mapping between RTL instructions and target instruction, however it is very low level in its highest form already as described below.

RTL is inspired by Lisp lists. It has both an internal form, made up of structures that point to other structures, and a textual form that is used in the machine descriptions and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form.

Each instruction in RTL form is represented as a nested tree of nodes, where each node has its *code* specifying the semantics, *mode* specifying the type the node operates on and *arguments*. Arguments are typed and type is known from *code*. `rtl.def` contains formal description of codes and lists of their types in the string form available to the compiler. This makes it easy to traverse RTL instructions.

For instance the following example is a RTL representation of 32bit integer plus operation:

```
(insn UID PREV NEXT (set (reg:SI 1))
                    (plus:SI (reg:SI 2) (reg:SI 3)))
```

`insn` node is an container with field `UID` containing unique identifier of instruction and `NEXT`, `PREV` of linked list of instructions.

`set` node is used to represent stores to the first operand (pseudo register 1) and the second operand is the actual expression. `SI` is mode representing 32bit integer (Single Integer).

RTL semantics is target independent making it possible to write common optimizers for all targets, however the syntax (set of allowed instructions) is target dependent. For instance i386 back-end describes conditional jump as:

```
(insn 56 13 57 (set (reg:CCGC 17 flags)
                    (compare:CCGC (reg:SI 61)
                                  (reg:SI 62))))

(jump_insn 57 56 33 (set (pc)
                         (if_then_else (ge (reg:CCGC 17 flags)
                                               (const_int 0))
                                       (label_ref 22)
                                       (pc))))
```

Closely describing the presence of flags register (register 17) on i386 architecture and split between compare and conditional jumps. Even the fact that compare instruction is not required to set carry flag is described by presence of `CCGC` mode. Same comparison on Alpha architecture looks different:

```
(insn 54 52 55 (set (reg:DI 80)
                    (lt:DI (reg:DI 78)
                           (reg:DI 79))))

(jump_insn 55 54 28 (set (pc)
                         (if_then_else (eq (reg:DI 80)
                                           (const_int 0))
                         (label_ref 17)
                         (pc))))
```

Representing the fact that Alpha does not have 32bit operations, and everything is done in 64bit (DI stands for Double Integer) and that general purpose registers are used to hold predicates for conditional jumps.

This close representation of instructions allows GCC to do lots of target specific optimizations (such as in i386 case use single comparison for multiple jump or conditional move instructions), and allows easy re-targeting of GCC without complicated peephole pass, but it brings difficulties to higher level optimizers, as analyzing RTL in few dozen possible forms is difficult. This explains the motivation for mid-level RTL, where for instance all conditionals do have the common form:

```
(jump_insn 14 13 33 (set (pc)
                         (if_then_else (ge (reg:SI 61)
                                           (reg:SI 62))
                                       (label_ref 22)
                         (pc))))
```

Full description of RTL and tree languages is present in [1].

4.2 Control Flow Graph

A control flow graph (CFG) is a data structure built on top of the intermediate code representation (RTL instruction chain or trees) abstracting the control flow behavior of compiled function. It is an oriented graph where nodes are basic blocks (chains of instructions allways executed in sequence) and edges represent possible control flows from one basic block to another.

GCC had multiple implementations of CFG in various optimization passes. For our project we needed to unify all representations and decided to base our work on the most modern one, implemented in 1997 by Richard Henderson for the liveness analysis and register allocator update.

Unlike the other implementations usually designed as arrays of basic blocks and bitmaps of edges, Richards implementation used linked lists to represent edges allowing us to modify CFG easily.

Full description of the CFG representation and overview of routines that we developed to maintain it easily is present in chapter 14 that we have submitted for inclusion into official GCC manual.

4.3 Optimization Passes

In this section we present short overview of optimization passes done in GNU compiler both before and after our changes. They are shown in figure 4.1. CSE refers to common subexpression elimination pass, GCSE/PRE refers to global[2] partial redundancy elimination pass, SSA refers to single static assignment form based optimizations, that are unfortunately not working in GCC at the moment.

For compactness the common cleanup passes run before or after pass, if enabled, are referred by letters. 'C' stands for CFG rebuild, 'J' stands for jump

[2]Passes operating only on instructions executed in sequence, basic blocks, will be refered as local. Passes operating on whole function body are global.

optimizer, 'J'' stands for our new CFG-cleanup pass replacing jump optimizer, 'D' stands for trivially dead instruction removal.

As one can easily see, we have dramatically reduced amount of CFG rebuilds. Our new jump optimizer implementation is faster, CFG and life info transparent which makes it possible to be run more times resulting in better code and less interference between optimization passes making development easier.

With tracer disabled, resulting compiler is faster than before, as CFG-cleanup overall consumes less time than old jump optimizer, liveness information is recomputed fewer times, as CFG is rebuilt. With tracer enabled, compiler is slower, since tracer performns lots of duplication and several optimization passes are run on more instructions than previously.

For description of individual passes, we shall refer reader to GCC manual [1] and chapter 8.

	before			after	
	Early passes			Early passes	
			C	High level branch pred.	
JC	Sibling Calls		J’	Sibling Calls	
JC	Exception Handling		CJC	Exception handling	J’
	Jump optimizer		J’	Loop test duplication	C
				Jump threading	
	SSA (disabled)			SSA (disabled)	
				Webizer	DJ’
DJC	NULL pointer elim.			NULL pointer elim.	DJ’
	Jump threading	J			
C	IF conversion			IF conversion	J’
	CSE	DJ		CSE	DJ’
C	NULL pointer elim.	DJ		NULL pointer elim.	DJ’
C	GCSE/PRE			GCSE/PRE	
	Copy/const prop.	DJ		Copy/const prop.	DJ’
				Register Coalescing	D
	CSE	J		CSE	J’
	Loop optimizer			Loop optimizer	CJ’
	Loop unrolling				
				Profiling	
				Branch prediction	
				Natural loop disc.	
				Profile estimation	
				Loop unswitching	
				Loop peeling	
				Loop unrolling	
			J’	Tracer	
				Webizer	DJ’
				RTL lowering	
	Jump threading	J			
	CSE	DJ		CSE	DJ’
	Profiling				
C	Branch prediction				
	Liveness anal.			Liveness anal.	J’
				Jump threading	
	Instruction combining			Instruction combining	DJ’
JC	Liveness anal.				
	Register allocation	JC		Register allocation	J’
	Crossjumping	J			
	Liveness anal.			Liveness anal.	
				Crossjumping	J’
	Late passes			Late passes	
C	Liveness anal.				
	Reg-stack	JC		Reg-stack	J’
	Crossjumping	J			
	Basic block reordering	J		Basic block reordering	
	Final passes			Final passes	
				Variable tracking	

Figure 4.1: RTL compilation passes

Chapter 5

The Profiler Changes

The effect of many optimizations can be maximized by identifying typical execution paths and hot spots. The typical case is branch prediction, where we need to decide which instruction will most likely follow. If we succeed, the program will run faster, because required instructions are pre-fetched and decoded, wrong prediction requires discarding of cacheline and therefore performance decrease. But this not the only case — almost any optimization will benefit from such information.

One way of achieving this is to run the program with some input data and record the execution path, or, more usually, the block or branch coverage.

Block coverage measures the number of times each basic block is entered, while *branch coverage* records the number of times each branch was taken. It's obvious that block coverage can be calculated from branch coverage, simply by summing the execution counts of incoming branches of given basic block.

The best results would be reached if we could run the program with the same actual input data, record the coverage, and then recompile the program with feedback from the measured data. However, this is in most cases impossible and it makes no sense to execute the same program on the same input data twice. Therefore, a different approach is used: a set of typical input data is created (called a trial data), coverage is measured for each run and all runs are merged to form a hybrid profile, which should represent coverage of a typical run. The final compilation takes this profile into account and if the trial data were well chosen, the compiled code should run faster.

Previously, gcc contained support for measuring block coverage, but it was broken. More recently, support for measuring branch coverage was introduced, but several fixes needed to be done — mainly thread-safe profiling and some kind of source code identification, so that measured data cannot be misinterpreted.

To support profiling, special code is inserted on edges of CFG that counts the number of times the edge was used. This is called instrumentation. Additionally some initialization and finalization code is added to record the result. This slows down the program execution a bit (about 5%).

Each module is associated with a file containing collected data. Simple utility named gcov can be used to display measured data, i.e., the coverage (hence the name).

In order to minimize the profiling overhead, not all edges are instrumented. A spanning tree is found in CFG, and only edges outside this spanning tree are instrumented. The execution counts for edges in the spanning tree can be easily calculated by inductive process.

During development, some code may be changed, and therefore profiling data involving this code must be invalidated. Obviously, there is no need to drop entire profile, but simply data for affected functions. The most important task is obviously detecting of changes.

In order to achieve this, we need some more specific information about instrumented functions, which must be stored in both object and file containing the data. This must obviously include offset and number of instrumented edges and function name, just to make sure that data are not garbled. To identify particular code of function, we add a cfg checksum. Because there is difference in code generated with and without instrumentation, we can't calculate checksum just from the RTL (otherwise all profiling data will be invalid when the code is finally compiled without instrumentation), therefore the checksum is calculated only from CFG structure, which is the same in both cases.

The data file consists of several blocks, which represent single runs. Each block consists of list of function names together with their checksums and counter values.

This approach has a few advantages:

- Different algorithms may be used to compute hybrid profile without need to recompile the binary and possibly removing old data. The current implementation simply sums counts of matching profiles.
- Because we keep all runs in the data file, we can use profiles for multiple versions of functions, e.g. due to different #definitions.

The biggest disadvantage is that the data file is growing after every run.

5.1 Thread Safe Profiling

The problems may occur if the profiled program is multi-threaded and the target machine cannot atomically add constants to memory — the results may get corrupted in this case. Additionally even on the machines supporting this operation we might run into problems if we used some optimizations on the profiling code (it seems reasonable to for example keep some of these counters in registers while we are in loop to reduce time overhead; if more threads did this simultaneously, the results would be unpredictable).

The classical solution to this problem — using locks — is not feasible in this case, as they are relatively time consuming (in order of tens of instructions; this itself could cause slowing down several times in short loops, plus there are unpleasant effects of necessary function calls on other optimizations).

We solve this by using per-thread counters. Each thread has its own set of counters created and initialized at its startup and their values are propagated into a central copy at its exit (using standard locking mechanism).

There are several technical problems associated with this solution:

- There are several thread implementations used. Fortunately, this problem was already solved (by creating a unified interface to them) due to some initialization issues, so we simply reused it.

- We must find where the counters for a given thread are located. We have considered several possibilities (reserving a register for it, adding a special parameter to functions, ...); from practical reasons (portability, ability to interact with non-profiled code), we had chosen to use standard thread-specific data facilities and to get the position of the counters on the beginning of each profiled function.

- It is impossible to recognize in compile-time whether threads will be used; therefore (to maintain performance in case when threads are not used), we had to check this in run-time.

In result, we have increased the overhead of profiling a bit — in programs that do not use threads we have additional check in each function for this plus counters are not stored in static table now, so we have some overhead on calculating the address; in threaded programs we also call function to get thread specific data address in the beginning of each function. The total overhead increased about 2.2 times without threads and about 3.1 times with them. Of course it is still possible to use old profiling code in case your program does not use threads (or you want to risk).

There are some problems still not solved (for example if program ends and some threads are still running, information from their counters is lost) and it would be certainly useful to decrease the overheads (moving reworked part of loop optimizer after profiling code generation helped a bit; moving remaining part (strength reduction) could help a lot).

Chapter 6

Branch Predictions

Although the profile feedback brings very accurate information about program behavior, it has one major drawback — an easiness to use. Majority of users are not willing to construct train runs for their programs and compile twice and in some cases (such as interactive programs, programs whose behavior depend on configuration or in embedded systems) this is even impossible to do.

An attractive alternative to feedback based profile is static branch estimation. Ball and Larus [3] describe a set of simple heuristics to predict conditional branch direction (taken or not taken) in about 75% of executed branches, that is a remarkable result, even when upper bound (the perfect predictor magically knowing in advance the actual program behavior) is about 91%.

Wu and Larus [4] extended this method to estimate conditional branch outcomes (probability that branch is taken) and propagate this information over intra-procedural control flow graph to compute expected number of executions of each basic block and they got satisfactory results predicting well the majority of hot spots in the program.

The original branch prediction code based on Ball and Larus heuristics was integrated into GCC as part of IA-64 porting project by Stan Cox and Jason Eckhart, but later it showed to be seriously broken in our experiments predicting loop branches in wrong direction resulting in about 55% successful predictions compared to 75% and it implemented just subset of predictions described in the paper.

6.1 Infrastructure

Goal of our implementation was to allow the front-end and early optimization participate on branch predicting process. This differed from the Ball and Larus approach that used final assembly exclusively for the heuristics most probably because the sources of actual compiler were not available to the researches. Because more information is available to compiler than one can easily derive from resulting code, we can predict better. We can also re-use the analyzers implemented in existing optimizer passes.

Our approach has three major passes. In the first pass, number of heuristics are tried on each conditional jump and if some of them predicts the direction of jump (taken or not taken), the information is stored into conditional jump instruction itself.

In the second pass we combine all the information gathered into single value, the *outcome* — estimated probability that given conditional jump will be taken.

In last pass we propagate the information into block coverage (see chapter 5) constructing equivalent data to the ones measured by the profiler. Optimization passes then may handle both estimated and feedback based profiles equivalently.

6.1.1 Implementing and Verifying Heuristics

We added a possibility to attach `REG_BR_PRED` note to each conditional branch in the RTL instruction stream. The note contains two values — identifier of the predictor algorithm for later analysis and the outcome the branch is supposed to have.

Each compilation pass run before profile estimation pass may freely predict any conditional jump in the code. For instance Ball and Larus suggest to predict a conditional jump around a loop as not taken. This heuristic comes from observation that most compilers copy test of `while(cond)`-type loops before the loop construct and loops tend to iterate. In our implementation the loop test duplication pass simply predicts each test copied resulting in better characteristics of this particular heuristics.

We found it difficult to predict whether given heuristic is good or not. To allow easy evaluation of new heuristics, we implemented tool `analyze_branches` to summarize information about behavior of all GCC heuristics based on real program behavior read from profile feedback. It can be used as follows:

1. compile program with instrumentation
2. execute program on train set
3. recompile program with profile feedback and debugging output (using `-db` command line option)
4. run `analyze_branches *.bp` on produced files to summarize information about heuristics

Following information is gathered about each of implemented prediction heuristics:

number of branches Number of branches in the compiled programs given heuristics apply to

dynamic hitrate Probability that direction of given branch in program is predicted correctly weighted by number of executions of each branch

predictability The hitrate of perfect predictor on branches predicted by given heuristics. Some heuristics predicts well predictable branches, while other predicts data dependent branches. This value represents upper bound of dynamic hitrate given heuristics can reach.

coverage Number of executions of branches predicted by given heuristics

It is obvious that heuristics in order to be successful should have dynamic hitrate and coverage as high as possible.

6.1.2 High Level Heuristics Support

Many run-time properties of program can be guessed from its source code. But some of properties are either completely lost or at least heavily mangled by translation into a low-level RTL representation. For example, usage of goto statement usually indicates some exceptional situation, like error handling; however, we cannot recognize jump created by goto statement from any other unconditional jump in the program. We can easily detect this statement during RTL generation, but using this information immediately would be hard and cumbersome.

To enable us to use these properties, we instead pass this information to later stages by emitting a special note (`NOTE_INSN_PREDICTION`) into insn stream. This note carries also additional information determining the kind of prediction and optionally some other attributes. In early stage (to avoid the information carried being influenced by optimizations) of the compilation we transform these notes into `REG_PREDICTION` notes placed directly on conditional jumps responsible for using the predicted branch (this property can be expressed as being at the end of the nearest basic block such that it dominates block containing the note, but is not postdominated by it).

6.1.3 Combining heuristics

Once all predictions are done, the prediction notes on each instruction must be combined into overall outcome of branch. When no heuristic applies to the branch, we predict it with outcome 50% and for purposes of debug output we refer this as "no prediction heuristic".

When single prediction applies, the situation is simple as well. Complicated is situation where multiple heuristics apply to single instruction. The method used by Ball and Larus paper is to use first matching heuristic in fixed priority order. While this method has been proved to work pretty well, it is difficult to set up the order of heuristics making development somewhat difficult. Wu and Larus paper suggests the use of Dempster Shaffer-theory of evidence:

We may threat predictions as hypothesis on theorem saying that given branch is taken. When there are two predictions with two different hitrates, p_1 and p_2 the combined probability can be computed as follows:

$$p = \frac{p_1 p_2}{p_1 p_2 + (1 - p_1)(1 - p_2)}$$

As the operation is associative, multiple predictions may be combined in order to get final outcome. This method is elegant, but it has been mentioned as problematic in followup articles [10], as it expects both predictors to be independent on each other that is, of course, not the case.

Our method uses a combination of both methods. For strong heuristics, such as loop heuristics, we use first match method and for weak heuristics we combine the outcomes. When at least one of strong heuristics applies to the branch, we use the first one and when no applies, we use Dempster Shaffer theory based method to combine values. This appears to bring the better from both worlds.

The perfect solution is not possible, as there are too many heuristics to measure outcomes for all the possible combinations. [10] suggests to use neural network based method to combine values, but we decided to not use it due to large complexity of solution giving relatively little extra value. GCC needs to be understood by its developers and we guess most of them are not experts for artificial intelligence.

Andreas Jaeger has set up an tester (`http://www.sue.de/~aj/SPEC`) that runs weekly SPEC2000 benchmark suite on current GCC tree and stores results of `analyze_branches` to the log file. We use those results as hitrate values for the prediction combining algorithm. `predict.def` describes all heuristics, their names, hitrates and prediction combination methods.

The exact implementation is described in algorithm 1.

Algorithm 1 Combining of branch predictions.

```
for all basic blocks b do
  if basic block ends by conditional jump then
    c ← 0.5
    p ← 0
    for all branch prediction notes n attached to the last instruction do
      if n.method = first match then
        if n.priority > p then
          b ← n.hitrate {hitrates measured experimentally by SPEC2000}
          p ← n.priority
        end if
      else
        c ← c * n.probability/(1 − c) * (1 − n.hitrate)
      end if
      if p ≠ 0 then
        c ← b
      end if
      predict branch edge by probability c and fall-thru edge by 1 − c
    end for
  else
    Predict all normal outgoing edges with equal probability and abnormal
    edges by probability 0.
  end if
end for
```

Table 6.1 contains table of heuristics we have implemented and parameters for the branch combining algorithm.

6.2 Heuristics implemented

6.2.1 High level heuristics

High level heuristics are the ones supplied by front-end and based on source language. Currently we have extended only the C and C++ front-ends to do so, but in future we plan to add heuristics to Java and Fortran front-end as well to handle for instance array bounds checking code or exception handling.

`goto`

Linux kernel commonly use `goto` statements to get into rarely executed paths of code. It can be argued that by good programmer, `goto` statement is usually

name	method	priority	hitrate
Loop iterations	F/M	10	computed
User supplied	F/M	9	98%
Loop branch	F/M	8	89%
Loop exit	F/M	7	90%/num_exits
Continue	D/S		56%
Noreturn	D/S		99%
Loop header	D/S		64%
Pointer	D/S		81%
Opcode positive	D/S		79%
Opcode nonequal	D/S		71%
Floating point opcode	D/S		90%
Call	D/S		70%
Goto	D/S		70%
Constant return	D/S		95%
Negative return	D/S		96%
NULL return	D/S		90%
Early return	D/S		67%

Table 6.1: Parameters of heuristics.

only used to solve exceptional situations, for example error handling/recovery. Branches ending with goto statements are predicted as not taken.

Continue

Continue statements create inner loops; these loops generally have different behavior than the real loops – they tend to roll less. Our experiments show that loop constructed by `continue` statement rolls approximately 1.1 times, while normal loops roll about 9 times. Disqualifying the loops constructed by continue statements from the loop heuristics allows us to increase hitrate of them.

Negative value return

Negative values are used in C to represent error stages, so functions usually do not take path leading to return statement with negative constant.

`NULL` return

Similar role as negative constants serve to integers, `NULL` serve to pointers, so path leading to return statement of `NULL` constant is predicted as not taken.

Constant return

Rest of the constants are also returned less frequently than computed values, so path leading to return statement with constant not predicted by any of the heuristics above is predicted as not taken.

User supplied

GCC extension is builtin function, `__builtin_expect` allowing programmer to explicitly state the outcome of given conditional.

6.2.2 Loop Based Heuristics

The edges belonging to natural loops in code are one of the best predictable edges in the program.

Ball and Larus paper suggests to predict all edges leaving loop as not taken using single loop heuristic based on simple observation that program must loop in order to do useful work and consume CPU time.

We have decided to split this single heuristic into multiple special cases in order to increase hitrate of common loop branches that in turn increase predicted amount of iterations in each loop allowing us to optimize more aggressively.

Loop Branch

The loop branch heuristic predicts last branch in the natural loop as taken in order to make loop rolling. We have split it from the loop exit heuristic below as we measured different behavior of this very latest branch in the loop compared to earlier exits, as this particular branch usually belongs to the loop test statement written by programmer, while the others are common caused by `break` statements in the loop body.

We bypass this heuristic for edges marked by continue heuristic as discussed earlier.

Loop Exit

Any early exit from natural loop is predicted as not taken. We divide the probability of this prediction by number of exits of loop to avoid prediction of number of iterations of loop to be dependent on number of exits, as this resulted in predicting very few iterations on common loops and producing poor code.

Loop Iterations

GCC contains code to analyze loop and compute number of iterations of well behaving loops with iteration variable and constant bounds. In such cases we use loop iterations heuristic specifying exact probability of the exit test.

6.2.3 Nonloop heuristics

Loop header

In case of while loops, the test conditional is duplicated in the front of loop to eliminate jump into the body and to allow more loop optimizations. The resulting branch is usually not taken, as loops tend to iterate. We predict branches created during loop exit test duplication as not taken.

Unlike Ball and Larus heuristics, we do not mark any branch around the loop, as the explicit conditionals are often limiting checks that allows us to increase hitrate of the predictor, at the expense of reducing its coverage.

Pointer

For `a` and `b` pointers, it we predict `a==b` and `a==NULL` to be false. The negations of the conditions are predicted to be true.

Positive

Most of values in program are positive, so for `a` integer, it we predict `a>0`, `a>=0`, `a>=1` and `a>=-1` as true. The negations of condition are predicted to be false.

Opcode nonequal

Similarly as for pointers, even for integers `a==b` is usually false. We disqualify from this heuristics comparison against `0` and `1`, as those constants are used for boolean expressions artificially lowering hitrate of the heuristics.

Floating point opcode

We predict floats to be non-equal and non-NaN.

Call

As suggested by Ball and Larus paper, we predict conditional jumping around call as taken, as calls are commonly used for error reporting.

Early Return

We predict branches causing function to return before reaching the last basic block not predicted by constant/negative/`NULL` return heuristics as taken. Ball and Larus suggest a reverse heuristic, claiming that early returns are used for error handling, but we take care of error handling by high level heuristic that allowed us to reverse a direction of the heuristic. Our heuristic can take care of fast paths through function, such as in hash table lookup, where single condition is usually guarding infrequently executed loop.

6.3 Estimating profile

Once probabilities of all edges in CFG are known, we produce estimated profile. In the acyclic tree, this is easily done by giving entry node frequency 1 and walking the tree down, computing frequencies of each node as sum of frequencies (once known) of its predecessors multiplied by the probability of given edge.

In the cyclic tree, the exact propagation is difficult and time consuming. We adopt simple algorithm described in [4] — to walk the loop tree from innermost to outermost, and at each level compute estimated number of iterations by determining a probability the loop-back edge will be reached when header block is executed. This can be done by simple modification of the algorithm above — when header of inner loop is reached, its estimated frequency is multiplied by already predicted number of iterations.

We found this algorithm fast and giving satisfactory results, as majority of functions in practice have reducible graph (ie. non-natural loops do not exist).

Chapter 7

Mid-level RTL

As already discussed in chapter 4, one of major GCC problems is that its intermediate language, RTL, is too low level for modern optimizations. As described in chapter 1 Jeff Law has been working on implementing mid-level RTL, a higher level counterpart to RTL. Once we finished basis of loop optimizer, we asked him how far he got in his efforts, as we desperately needed the mid-level RTL to progress on loop optimizers. He told us that he had not been able to work on it and asked us whether we can attempt to implement it, so we decided to do so.

Mid-level RTL has the same representation and semantics as low-level RTL, so all existing optimizers can use it directly. In fact it is implemented simply as yet another GCC target having its own machine description `midrtl.md`. The instruction set is highly regular RISC-style allowing optimizers to easily analyze and generate it.

We have virtualized all interfaces in dealing with RTL and we can switch the machine description during the code generation process that is done after second CSE pass. Instruction splitting pass is used to regenerate RTL chain in its low level representation and compilation continues.

We must mention, that our work is not quite finished in this respect, since all existing target descriptions need modifications in order to handle function call sequences correctly (we were unable to adapt current interface to the new form). We have updated i386 machine description and disabled mid-level RTL construction in all other targets for now. We plan to work with target machine description maintainers on getting majority of them updated while integrating our change to the official tree.

All our other changes are designed in a way that they do not require midlevel RTL, but benefit from it when available.

Chapter 8

Optimization passes

8.1 CFG Cleanup Pass

GCC jump optimization pass was implemented using simplistic RTL instruction chain analysis and it disturbed control flow graph completely during optimizing.

Since this pass needs to be run many times during compilation and the implementation was somewhat rotten, we have decided to re-implement it completely. In fact the decision predates the project itself and was the first step done in the direction of pushing CFG into compiler. In the first attempt to re implement the pass we failed miserably hitting all sorts of limitations of the control flow graph manipulation code giving us the main motivation for goals of our project.

With important effort spent on the CFG manipulation infrastructure we have succeeded to re-implement all features of old jump optimizer in stronger and faster way. We perform transformations described bellow in the loop until the code stabilizes (ie. no transformation matches after searching the whole flow graph).

8.1.1 Edge Forwarding

The edges whose destination is a "forwarder block", where the forwarder block is an empty block or block containing only the unconditional jump instruction, are forwarded to the destination of successor edge of the block.

We have implemented new function to redirect edges in the flow graph and update underlying RTL instruction, that also attempts to simplify branch instruction in the source basic block. When all edges of the computed jump or conditional jump are forwarded to the same destination, the jump instruction is replaced by unconditional jump or removed entirely.

This function is now used in many other places in compiler and this single transformation supersets number of special subcases handled by old jump optimizer code, such as computed jump or table-jump removal.

8.1.2 Jump Threading

Jump threading is an optimization similar to edge forwarding but taking important special case of conditional jump targeting conditional jump with condition whose value can be determined from the condition of first jump. For instance in following code:

if $a < 0$ **then**
  code not clobbering a
end if
if $a >= 0$ **then**
  code
end if

Jump threading adds missing `else`.

We have implemented jump threading by running CSE over the two basic blocks as if they were a single block. When CSE is able to determine constantness of resulting block we attempt to prove that second basic block has no side effect and forward the edge on success.

8.1.3 Cross Jumping

Cross jumping, also known as tail merging, is a optimization eliminating duplicates in the code. It is identifying sequences of identical code at the end of basic blocks targeting same destination and unifying it. In the simple form it can be used to transform:

if $a < 0$ **then**
  code
  $a \leftarrow 0$
else
  different code
  $a \leftarrow 0$
end if

into:

if $a < 0$ **then**
  code
else
  different code
end if
$a \leftarrow 0$

We implemented generalized form that also identifies control flow transfer instructions that have equivalent effect. This allows, for instance, to transform:

if $a < 0$ **then**
  code
  if $b < 0$ **then**
    $b \leftarrow 0$
  end if
else
  different code
  if $b < 0$ **then**
    $b \leftarrow 0$
  end if
end if

into:

if $a < 0$ **then**
  code
else
  different code
end if
if $b < 0$ **then**
  $b \leftarrow 0$
end if

Such commonalities are surprisingly frequent in the real world code, as they are easily created by function inlining. Also our tracer optimizer increases greatly the amount of cross jumping opportunities. The overall code reduction caused by this optimization is about 10%.

8.1.4 Unreachable Code Removal

We perform simple depth first search through the graph marking all nodes reachable from entry block and removing all unmarked basic blocks.

8.1.5 Unnecessary Code Labels Removal

Code labels that are no longer reached by branch edges are either removed, or converted into DELETED_LABEL notes in case they are still referenced from data segment.

8.1.6 Conditional Jump around Jump Simplification

This transformation simplifies:

if $a < 0$ **then**
  goto skip
end if
goto branch
skip:

Into:

if $a >= 0$ **then**
  goto branch
end if

8.1.7 Basic Block Merging

In case basic block has a single exit edge that targets other basic block with a single entry edge the basic blocks can be merged. We perform code reordering when needed.

8.1.8 Liveness, CFG and Profile Maintenance

Most importantly our new implementation is transparent to both liveness information and profile. It allows us to run the pass more often producing better code. Also our pass is faster than old implementation, since it does not have to examine RTL chain in detail and can recognize the optimization opportunities on the shape of control flow graph easily.

8.2 Loop Optimizer Rewrite

We were unable to adapt the old loop optimizer in GCC to new CFG representation. The reasons were:

- It is quite big (14939 lines) and complicated piece of code.
- It mostly ignores basic block structure in unrolling part, thus making it very hard to maintain the information.
- Its design is not very good; to locate loops, it uses information passed from frontend (notes emitted into insn stream). During former optimization passes, this information often gets mangled, thus preventing the loop optimizer from optimizing the loop.

We have decided to rewrite the loop optimizer from scratch. To keep the functionality, we are slowly replacing it by parts. Till now we have already replaced loop unrolling part and used the created infrastructure to extend it by other natural loop transformations (loop peeling and unswitching).

8.3 Loop Data Structure Changes

The old loop optimizer kept information about the loops in a rooted forest (nodes corresponding to loops, edges to subloop/ superloop relation). Basic blocks belonging to a loop were recorded in a bitmap indexed by basic block numbers. This structure was not flexible enough for our purposes – basic block numbers sequentially number basic blocks along insn chain and therefore may change during CFG transformations (a bit unpleasant design choice, making it hard to index anything by them; we are working on changing this). It also was not very efficient for some common operations (for example finding body of a loop required scanning every basic block in the function, finding innermost loop the basic block belonged to was done by testing membership in all the loops).

We decided to change the loop data structure. In our design we came out from the following assumptions:

- The loop tree is usually shallow – you will hardly find practical example of program with 10 nested loops.
- Loops are not created or destroyed too often.
- On the other hand, we relatively frequently change a set of basic blocks that belong to loops.

We have divided the information about loops into 2 parts; one of them is stored in CFG, the other one remains in the loop tree. In the loop tree we store informations describing the loop (pointer to its header and latch, its size), while in CFG we have for each basic block a pointer to the innermost loop it belongs to. Furthermore, we have added a dummy loop containing whole function as a root of the loop tree (making some functions a bit simpler) and for each node of the tree an array of pointers to all its superloops.

With this structure, we are able to provide following operations:

- CFG manipulation in $O(1)$.

- Queries for membership in loop, innermost containing loop and subloop/ superloop relationship in $O(1)$.

- Loop structure changes in $O(\text{tree depth})$.

It is still not straightforward to find a body of the loop; we support this fundamental operation by exploiting natural loop properties (for definition of a natural loop see following section). From the definition it follows that we can enumerate the loop by a simple DFS from the latch backwards along the edges, stopping on the loop header; this is linear in loop size and fast enough for our purposes.

Compared to previous implementation, we only lack possibility of having loops with more than one latch. This is not a problem, as we are always able to eliminate them (as explained in the following section).

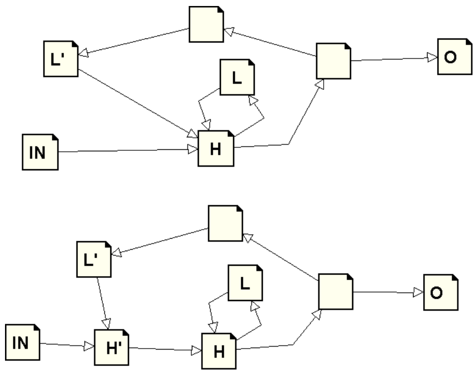
8.4 Natural Loop Discovery Changes

Natural loop is a maximal (in inclusion) set of basic blocks such that it contains a special one (header) that dominates all of them and is backreachable from them. Predecessors of header inside loop are called latches. If there is a single predecessor of header outside the loop, we call it a preheader.

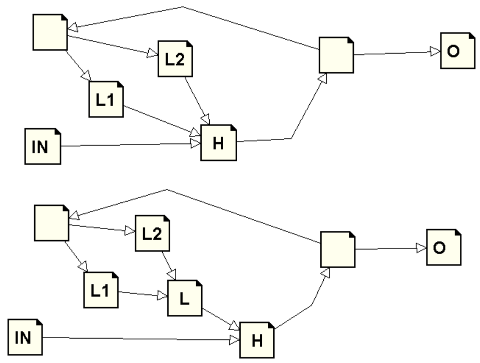
Natural loop discovery code (based on ...) was provided by Michael Hayes (but it was not used in loop optimizer yet); we only had to adapt it for a new loop infrastructure. In following paragraphs we address some issues concerning the change.

As mentioned earlier, our data structure does not support loops with more than one latch basic block. This is not a great problem, as such loops are somewhat nonstandard and we would have to bring them into a canonical shape anyway. Loop with more latches can occur from the following reasons:

- Presence of the inner loop with the shared header. The correct way to eliminate this case is to create a new empty header for the outer loop.



- Several branches inside loop ending in the header; we eliminate this by creating a new empty latch.



It is relatively hard to distinguish these cases (but they are not too common anyway, so it is not too important). If we have profile information, we can differ between them; while in the latter case frequencies of the latch edges are similar, in the former case latch edge of inner loop tends to be much more frequent. Otherwise we simply use the second way.

In addition, we have also provided some functions to bring loops into even more canonical shape (creating preheaders; forcing preheaders/latches to have only a single outgoing edge to the loop header) in that it is much easier to keep the loop structure up to date during transformations like loop copying etc.

8.5 Loop Optimizer

We started our work on new loop optimizer by replacing the part responsible for most structural changes – the unroller. We have postponed work on other optimizations (induction variable elimination, strength reduction) until we have mid-level RTL and SSA form that should make the work much easier.

We have implemented basic functions for CFG manipulation able to update loop structure (and also to keep dominator tree up-to-date, which is important for some of our optimizations).

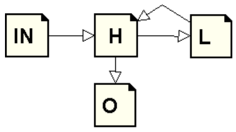
Using them, we have created several useful functions for loop structure manipulation (copying of loop body to the edge coming into its header – operation common for both loop unrolling and peeling, removing the branch of if condition inside loop – for unswitching and clever loop unrolling, loopification of a part of the CFG, creating a canonical description of simple (in sense that we are able to determine number of their iterations) loops).

We are able to keep profiling information consistent in most of the times (the only place where we have problem is removing branches; it is hard to recalculate frequencies precisely afterwards, so we only use some approximation).

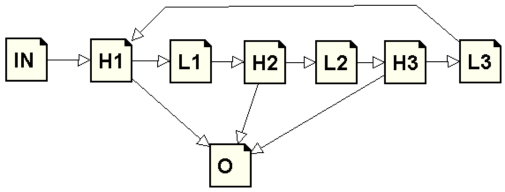
8.5.1 Loop Unrolling

By loop unrolling we mean copying the loop body several times. The result is that in a loop that iterates often enough we create longer run of instructions that are executed sequentially (without jumps). Such code is often executed faster.

```
while (1)
 {
  puts("Header");
  if (should_end ())
   break;
  puts("Latch");
 }
```




```
while (1)
 {
  puts("Header");
  if (should_end ())
   break;
  puts("Latch");
  puts("Header");
  if (should_end ())
   break;
  puts("Latch");
  puts("Header");
  if (should_end ())
   break;
  puts("Latch");
 }
```



With the described machinery, unrolling is quite easy – it suffices to copy loop body on the edge from latch to header.

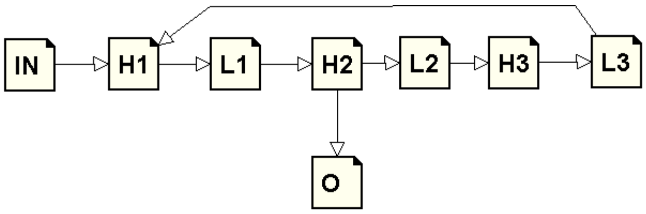
This stupid loop unrolling is not always a win; it makes code to grow, which may cause problems with caches, making the program to run slower. To avoid this, we only unroll small loops. With profile informations, we only unroll loops that are expected to iterate often enough and also disable the optimization on places that do not execute often enough.

We have also written some more clever variants of the basic loop unrolling scheme:

```
for (i = 0; i < 10; i++)
 printf ("%d\n", i);
```

↓

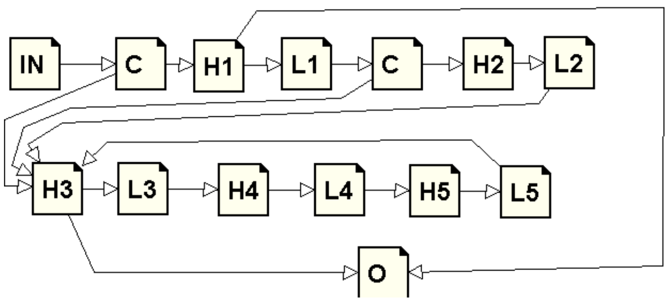
```
i = 0;
while (1)
 {
  printf ("%d\n", i++);
  if (i>=10)
   break;
  printf ("%d\n", i++);
  printf ("%d\n", i++);
 }
```




```
for (i = 0; i < k; i++)
 printf ("%d\n", i);
```

↓

```
i = 0;
m = (k - i) % 3;
if (!m--)
 goto beg;
if (i>=k)
 goto end;
printf ("%d\n", i++);
if (!m--)
 goto beg;
printf ("%d\n", i++);
beg:
while (i < k)
 {
  printf ("%d\n", i++);
  printf ("%d\n", i++);
  printf ("%d\n", i++);
 }
end:;
```



Unrolling Loop With Known Number Of Iterations

If we are able to determine the number of loop iterations exactly, we can either peel the loop completely (removing the loop) if it is sufficiently small, or at least determine precisely the exit that will be taken out of the loop (by counting the number of iterations modulo the number of unrollings). We can then remove exit edges that we know that are not used.

Unrolling Loop With Number Of Iterations Known At Runtime

Often it happens that we are not able to count number of iterations in time of compilation, but we are able to count it in runtime. Then we can add code to ensure that number of iterations is divisible by number of unrollings (by peeling loop several times and adding conditions to check that correct number of iterations is remaining, then unroll the loop and remove exit tests from all but one copy of body.

This is not the whole story, as in this case we have to count with some degenerated instances. The easy one is that the loop may not roll at all; this we can check rather easily. More difficulties arise from the fact that we also have to check for overflows. Fortunately, if the number of times we unroll the loop is a power of 2, this will not cause the problems (as overflows only occur in case the the final check is for equality and range of integer types is power of 2, therefore modulo number of unrollings the result is the same).

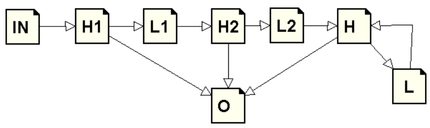
In both of these cases we benefit not only from greater sequentiality of code, but also from removing the unnecessary checks and better possibilities for further optimizations.

8.5.2 Loop Peeling

```
while (1)
 {
  puts("Header");
  if (should_end ())
   break;
  puts("Latch");
 }
```

$\rightarrow$

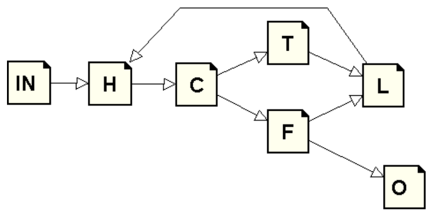
```
puts("Header");
if (should_end ())
 break;
puts("Latch");
puts("Header");
if (should_end ())
 break;
puts("Latch");
while (1)
 {
  puts("Header");
  if (should_end ())
   break;
  puts("Latch");
 }
```



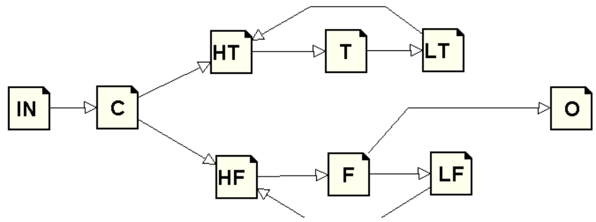
Peeling is similar to unrolling, but instead of copying loop to latch edge, we copy it to preheader edge, thus decreasing the number of loop iterations. Peeling is used if number of loop unrollings is small; the resulting code is again more sequential.

8.5.3 Loop Unswitching

```
i=10;
while (1)
 {
  if (flag)
   puts("True");
  else
   {
    puts("False");
    if (!i--)
     break;
   }
```




```
i=10;
if (flag)
 while (1)
  puts("True");
else
 while (1)
  {
   puts("False");
   if (!i--)
    break;
  }
}
```



Loop unswitching is used in case when some condition tested inside loop is constant inside it. Then we can move the condition out of the loop and create the copy of the loop. Then in the original loop we can expect the condition to be true (and remove the corresponding branch) and in the copy to be false (and again remove the branch).

We must be a bit careful here, as if we had several opportunities for unswitching inside a single loop, the code size could grow exponentially in their number. Therefore, we must have some threshold over that we will not go.

Other interesting case is when we test for the same condition several times. We can then use our knowledge of which branch we are in to just eliminate the incorrect path; the only small problem associated with this is that we could create unreachable blocks, or even cancel the loop. Both of these cases are hard to handle, so we just revert to former case (it does not occur too often, and the following optimization passes will get rid of that much apparently unreachable code anyway).

8.6 ET-Trees

We used ET-trees for maintaining dominance information in loop optimizer. *ET-tree* is a data structure for representing trees. It offers logarithmic time for standard tree operations (insert, remove, link, cut) and poly-logarithmic time for finding common ancestor of two nodes.

The tree is represented by a sequence of symbols obtained by calling algorithm 2 on root node.

Algorithm 2 dfs (node)

$s \leftarrow node$
for all children c **do**
  $s \leftarrow$ concat(s, dfs(c), $node$)
end for
return s

Such sequences are called Eulerian Tours, hence the name of the structure. For example, for tree on figure 8.1, the corresponding ET-sequence is `1,2,1,3,4,5,4,6,4,3,1`.

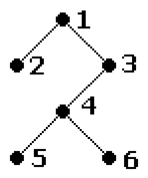


Figure 8.1: A tree with ET-sequence 1,2,1,3,4,5,4,6,4,3,1

Because it's inefficient to maintain these sequences as strings, slightly different approach is used — the sequence itself is represented by a binary tree, where the represented sequence is obtained by reading the values of the nodes from left to right.

For example, two possible encodings of the above sequence are shown in figure 8.2.

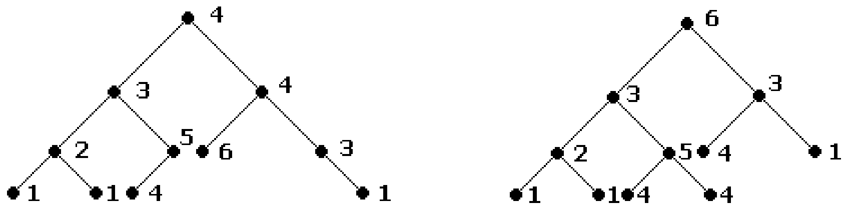


Figure 8.2: Two possible encodings of the above sequence.

In our implementation, we have chosen splay-trees as the sequence representation, with some modifications:

1. All occurrences of the same node are linked from left to right, this allows faster implementation of `remove`.

2. Because node values can be of any type, a hashtable is used to convert node values to appropriate internal representation.

There are two data types for representing nodes — `et_forest_occurrence` and `et_forest_node`. `et_forest_occurrence` represents the occurrences of the nodes in tree (therefore, these nodes form the splay-tree). It contains informations for the splay-tree (parent node, left son, right son), pointer to the next occurrence of the same node, and number of nodes in left and right subtree (this is used for comparison whether an occurrence is before another one, or not). `et_forest_node` represents each node. It contains pointers to the first and last occurrences of the node and the node value. The `et_forest_t` is then simply a hashtable converting node value to appropriate `et_forest_node`.

Implementation of of tree operations

1. Insertion of new node is trivial — we simply create a node with one occurrence.

2. Insertion of an edge — we splay first occurrences of both nodes, create new occurrence of parent node and link them together as shown in figure 8.3.

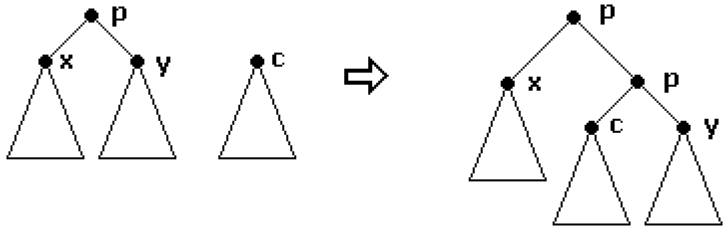


Figure 8.3: Insertion of edge (p,c).

3. Removal of a node — We simply remove edges to child nodes and possibly the edge to parent node and then dispose the node with single occurrence.

4. Removal of an edge — Because we know that the occurrences before the first occurrence of child node and after the last occurrence of child node belongs to parent node, we cut the sequence using two splays and one join, removing one occurrence of parent node, as shown in figure 8.4.

5. Nearest common ancestor of two nodes is calculated by algorithm 3.

6. Splay trees are well known and therefore not covered here.

8.7 Register Coalescing Pass

We found another irritating defect of GCC register allocation pass — the fact that it is not able to eliminate redundant register to register copies. The copies are commonly created as a result of global common subexpression elimination (GCSE) pass and resulting code is often slower than when GCSE is disabled. In our efforts we have fixed some defects of GCSE resulting in more changes to be done making this problem serious. This was the main motivation to implement simple register coalescing pass.

The pass first scans function and builds conflict graph of pseudo registers (vertexes of graph are registers and two registers are connected by edge when

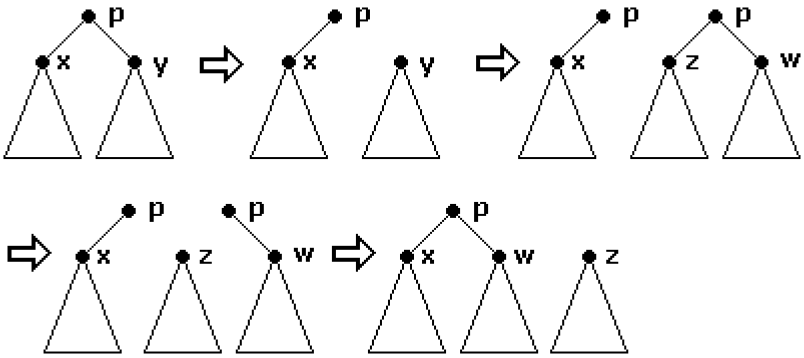


Figure 8.4: Deletion of edge (p,c).

Algorithm 3 nearest common ancestor (v1, v2)

$p1 \leftarrow$ position of first occurrence of $v1$
$p2 \leftarrow$ position of first occurrence of $v2$
if $p1 < p2$ **then**
    $max \leftarrow p2$
    $candidate \leftarrow v1$
else
    $max \leftarrow p1$
    $candidate \leftarrow v2$
end if
while position of last occurrence of $candidate < max$ **do**
    $candidate \leftarrow parent(candidate)$
end while
return $candidate$

they are live together, so they cannot reside both in a single register). Later the program is rescanned and each register-to-register copy operation is considered. When there is no edge between both operands of the copy, the registers are marked for coalescing and conflict graph is updated. Finally the instruction chain is updated to represent the coalesced form.

We have found this pass to be relatively effective (eliminating all negative effects of GCSE), however we are not sure whether it will be merged to mainline GCC, as new register allocation branch already contains register allocation able to handle the situation in question. We are considering extending the pass to replace current GCC regmove pass, that is doing essentially the same, but its purpose is to convert RTL chain into form representable in 2 address instruction set (if target architecture has one), when the destination of operation is equivalent to its source.

First we need to discuss with register allocation developers what level this transformation should be done on. Also it is possible that the new register allocation will not be merged into mainline tree until the next release, in that case we will attempt to merge our pass as a temporary solution.

8.8 Webizer

Modern register allocation algorithms are usually not based on pseudo registers, like the GCC one, since pseudo registers representing variables may be reused for no real purpose. For instance many programmers are using `i` in the single function as index counter in many different loops. It is stupid to constraint register allocator to allocate single hardware register for all occurrences of variable `i`.

Solution to this problem is to split each pseudo register into one or more *webs*. The name web comes from DU data-flow representation. In this form, each definition of register has an attached linked list pointing to all instructions possibly using the value (Definition Use chains). The webs can be constructed by unifying all definitions having the same uses (as when instruction uses value from multiple possible definitions, all definitions must come to the same register).

We have implemented simple pass that constructs DU chains, computes webs and when single pseudo has attached multiple webs, it creates new pseudo register, so at the end there is $1-1$ correspondence between pseudo registers and webs.

This allows register allocation to be stronger without additional modifications. In addition this makes some other optimization passes stronger. For instance the following piece of code

$a \leftarrow a+1$
$b[a] \leftarrow 1$
$a \leftarrow a+1$
$b[a] \leftarrow 1$
$a \leftarrow a+1$
$b[a] \leftarrow 1$
$a \leftarrow a+1$

after webizing looks like:

$a \leftarrow a+1$
$b[a] \leftarrow 1$
$a_2 \leftarrow a+1$
$b[a_2] \leftarrow 1$
$a_3 \leftarrow a_2+1$
$b[a_3] \leftarrow 1$
$a \leftarrow a_3+1$

and GCC Common Subexpression Elimination pass converts it into:

$a \leftarrow a+1$
$b[a] \leftarrow 1$
$b[a+1] \leftarrow 1$
$b[a+2] \leftarrow 1$
$a \leftarrow a+3$

when offsetted addressing is available. We run Webizer twice — once in early compilation stages and once after tracing and loop unrolling, so we do not need to implement special purpose pass to split induction variables.

8.9 Software Trace Cache

8.9.1 Theory

We use a greedy algorithm similar to *Software Trace Cache* [7] for basic block reordering. It reorders basic blocks to minimize the number of branches and instruction cache misses and does also a limited amount of code duplication during the process to avoid jumps. The effect is the increase of code sequentiality and thus also the increase of the speed of code execution.

The algorithm is based on profile information. It means that the results depend on the representativity of the training inputs. Thus the code can be worse for some input sets. If you have not profiled the program the algorithm uses the probabilities of branches estimated by a set of heuristics instead of profile information.

We minimize the number of (executed) branches by constructing the most popular sequences of basic blocks (the traces). To do that we are reordering the most probable successor after its predecessor. If the most probable successor has been already laid out we duplicate it to avoid jump if we are not duplicating a large chunk of code. A trivial example of duplication is copying the return instruction instead of jumping to it.

8.9.2 Implementation

The implementation does only intra-procedural optimizations since inter-procedural optimizations are out of reach of current GCC.

The algorithm constructs traces in several rounds. The construction of traces starts from the selected starting basic blocks (seeds). The seeds for the first round are the entry points of the function. When there are at least two seeds the seed with the highest execution frequency is selected first (this places the most frequent traces near each other to avoid instruction cache misses). Then the algorithm constructs a trace from the seed by following the most probable successor. Finally it connects the traces.

As we mentioned before the most frequently executed seed is selected. But there are some exceptions: The seeds whose predecessors already are in any trace or the predecessor edge is a DFS back edge are selected before any other seeds. This heuristics tries to create longer traces than the algorithm would create without this heuristics. In figure 8.5, the blocks H, A, B and E are already in a trace, the seeds are C and D. The seed C will be selected first because all its predecessors are already in some trace, even if the execution frequency of D is higher.

Given a basic block, the algorithm follows the most probable successor from the set of all unvisited (ie. not in any trace) successors and visited successors that can be copied. The decision whether the visited block can be copied is done by simulation of the trace building and counting the number of bytes that would be copied. The block can be copied only if the size that would be copied is lower than a threshold. The threshold depends on the size of the jump instruction and on the frequency of the block.

While constructing the trace the algorithm uses two parameters: *Branch Threshold* and *Exec Threshold*. These parameters are used while scanning the successors of the given block. If an edge to a successor of the actual basic block

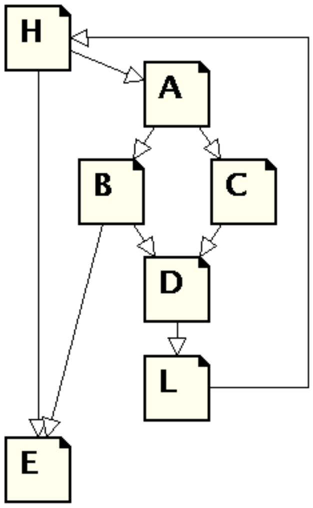


Figure 8.5: Selecting seed ‘C’ before ‘D’

is lower than Branch Threshold or the frequency of the successor is lower than Exec Threshold the successor will be the seed in one of the next rounds. Each round has the Branch Threshold and Exec Threshold less restrictive than the previous one. The last round has to have these parameters set to zero so that the last round could ”pick up” remaining blocks. The successors that has not been selected as a continuation of the trace and has not been ”sent to” next rounds will be added to seeds of current round and the secondary traces will start in them.

There are several cases of the selected successor's state (the action for the first matching state will be done):

- The duplicate of the successor is in this trace. We have found the loop in original flow graph. The algorithm terminates the construction of the trace.
- The successor has been visited in this trace. We have found the loop that is rotated if it is profitable. The construction of the trace is terminated.
- The successor has been visited (not in this trace). It is duplicated and the duplicate is added to the end of the trace.
- Otherwise. It is added to the end of the trace.

Figure 8.6 shows an example of the flow graph. We use an Exec Threshold of 50 and Branch Th. of 30% in the first round. Given a seed A we follow the edge to H1, and H2 is sent to next round because of Branch Th. We continue to L1 from H1, B1 is sent to next round. The best (and the only) successor from L1 is H1. We have found a loop that has enough iterations so we rotate it and terminate the trace. There are no more seeds in this round so the next round starts. We use Branch Th. of 0% and Exec Th. of 0 so this is the last round. The seed B1 has higher exec count than H2 so the next trace starts in B1. We follow the edge to R and terminate the trace because the block R is the exit block of the function (in GCC representation, there is an edge from R to EXIT_BLOCK). Next trace starts in H2 and continues to L2, B2 will be a seed for secondary trace. In L2, we find a loop that does not have enough iterations so we just terminate the trace. The last trace starts in B2. The successor R is already in some trace so we check whether we can duplicate it and realize that we can. So we duplicate it. Since it is the exit block of function, we terminate the trace. There are no more unvisited blocks so the trace building is done.

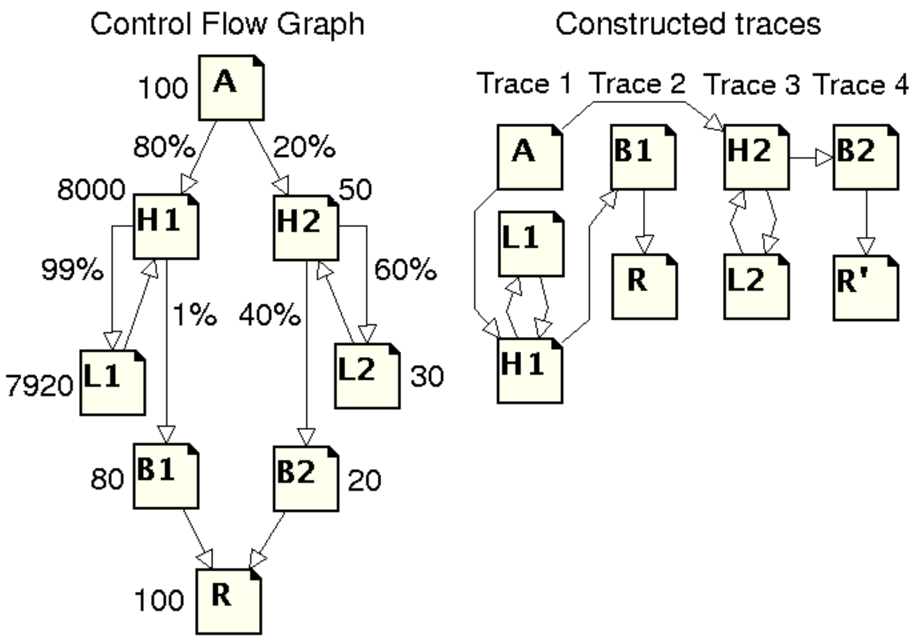


Figure 8.6: Trace building

Finally we connect the traces. We start with the first trace (that contains the entry block of function). Given a chunk of connected traces we prolong it. If there is an edge from the last block of the chunk to the first block of some trace that trace will be added to the end of the chunk. If there is no such trace the next unconnected trace is added to the chunk.

For the graph in figure 8.6, the traces happen to be connected in the same order as they were found.

8.10 Tracer

Tracer [2] performs the tail duplication which is needed for superblock formation. Usually it is used for scheduling but we have not tested it because GCC does not have general Superblock Scheduler (GCC has it only for IA-64) but we plan to generalize it in the future. We do not suppose that the scheduling will be profitable for Athlon (where we tested everything) because Athlon itself does the scheduling. On Athlon, we use Tracer to improve the effect of other optimizations. We do not need to worry about the code growth too much because Cross-jumping removes the unneeded tail duplicates when optimizations are not performed. The simple example how tracer helps other optimizations is the following code. Tracer makes possible for CSE to remove the second condition.

Example: Code where Tracer helps other optimizations.

```
if condition then
  t1 ()
else
  f1 ()
end if
c ()
if condition then
  t2 ()
else
  f2 ()
end if
```

The start trace finding from the basic block that is not in any trace yet and has the highest execution frequency. The Fibonacci heap is used for fast finding of such a block. The trace is then grown backwards and forwards using mutual most likely heuristic. The heuristic requires that for block A that is followed by block B in the trace, A must be B's most likely predecessor and B must be A's most likely successor. While the trace is growing backwards the backward growing can also stop if the probability of the most likely predecessor is lower than `MIN_BRANCH_RATIO` and while the trace is growing forwards the growing can also stop if the probability of the most likely successor is lower than `MIN_BRANCH_PROBABILITY`.

When we have a trace we perform tail duplication. We visit the blocks on the trace 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.

The trace finding and tail duplication is repeated until we cover the ratio of `DYNAMIC_COVERAGE` of the number of dynamic instructions (number of executed instructions in the function). The process also stops when the number of instructions exceeds `MAXIMAL_CODE_GROWTH` multiplied by the original number of instructions because we have to keep code growth and compiler resources under control although the unneeded duplicates will be eliminated later in Cross-jumping. The remaining basic blocks are the traces of one basic block.

Finally the traces are connected into one instruction stream. When there is an edge from the last basic block of one trace to the beginning of another these traces will be connected. Remaining chunks are connected in the order of appearance. We do not need to worry how well the traces will be connected because Basic Block Reordering will order the blocks so that the code will be (almost) as fast as possible.

Figure 8.7.a shows the original flow graph for an example code on page 57. We start finding the trace in basic block S since it has the highest execution

frequency. The trace is grown forwards by basic blocks C1, T1, C2, T2 and E. While visiting the blocks on the trace in trace order we duplicate block C2 because it has two predecessors. Basic block T2 has two predecessors now because of duplication of C2 (see figure 8.7.b) so we duplicate it too (see figure 8.7.c). Finally we duplicate basic block E too. Figure 8.7.d shows the result after tail duplication of the first trace. We start finding next trace in E. The trace grows backwards by basic block F2. It cannot grow by C2 because F2 is not the most likely successor of C2. The basic block E is then duplicated. The finding of next trace starts in basic block F1 and grows forwards by blocks C2, T2 and E. All basic blocks in this trace have one predecessors so no basic blocks are duplicated now. There are no more basic blocks left. The first trace is S, C1, T1, C2', T2', E', the second one is F2, E" and the last one is F1, C2, T2, E.

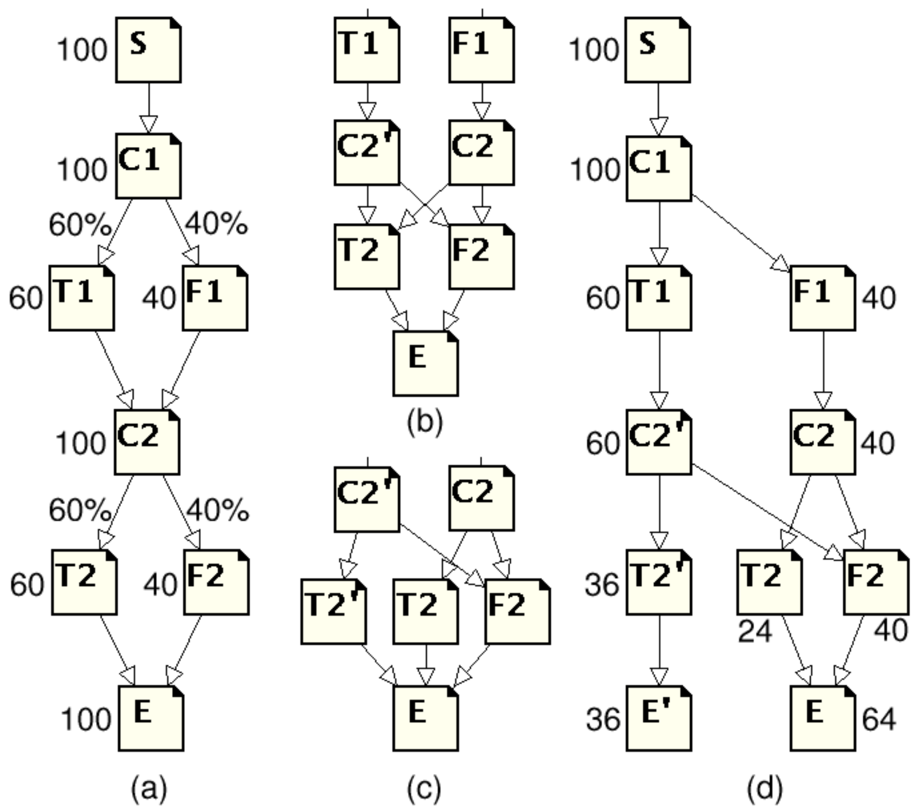


Figure 8.7: (a) The original flow graph for example code on page 57. (b) A part of flow graph after duplicating C2. (c) A part of flow graph after duplicating T2. (d) The flow graph after tail duplicating the first trace.

8.11 Code Alignment

The modern CPUs are usually getting instruction decoder stalls when branch targets are near code cache-line boundary. To avoid the problem, the targets of common branches and bodies of loops are recommended to be aligned to start at new cache-line boundary. Also when proceeding a code block that is rarely executed, targeting the branch just before cache-line boundary is wasteful concerning the code cache pollution.

Old GCC contained code to align all loops found in the code, all function bodies and all code following unconditional jumps to specified values according to target machine description. This strategy is however somewhat wasteful. For instance AMD Athlon chip recommends to align to 32byte boundary wasting 16 bytes up to 20% of code at the average.

We have implemented new pass that uses profile to carefully place alignments. We use the following set of conditions:

- in case basic block is not reached via fallthru edge check:
  - basic block is likely to be executed at least once
  - the sum of branch frequencies is very high (1/10th of maximum inside function) or the previous basic block is unlikely executed

  If both conditions hold, the basic block is aligned by the same alignment as old code did for instruction following unconditional jump.
- in case basic block is reached by both fallthru and branch edge check:
  - basic block is in the hot spot of the program
  - the sum of frequencies of branch edges is at least 5 times higher than fallthru edge frequency
  - basic block is in the very hot spot in the function

  If all the conditions are met, the basic block is aligned by loop alignment.

We have found that our code limits code growth to 5% while maintaining approximately the same performance in all benchmarks.

8.12 Miscellaneous Changes to Existing Optimizers

We have modified following existing optimizers to use profile feedback:

register referencing Register referencing pass decides what register class the given pseudo register should be allocated in. This is done by combining preferences of all instructions mentioning the given register. We now weight the preferences by expected frequencies of basic blocks.

register allocation GCC register allocator simply assigns each pseudo register a priority and then allocates registers to pseudo registers in the priority order giving each pseudo register the first register available in its preferred class and then the first available register in the alternate class if it did not succeed in the preferred class.

We have modified the code to compute pseudo register priority to use frequencies instead of simple loop depth heuristics and have important success with this change speeding up the resulting code by about 0.6%.

final pass Final pass outputs code alignment recommended for function body by target machine description. We omit the alignment, where allowed, for functions not executed in the training run when profile feedback is available.

register-stack conversion Register stack conversion pass has property that it always produces code optimal only for one entry path to each basic block. We have modified its traversal to take into account frequencies in order to optimize the most frequent paths.

8.13 Double test conversion pass

Double test conversion pass is a pass discovering conditionals in the form of `(cond1||cond2)` or `(cond1&&cond3)` and contained in the same superblock and attempts to construct equivalent sequences without branch instruction. In the trivial case `a||b` this can be done by converting into arithmetics `a|b` that in i386 instruction set even compiles into equivalently long code in case both variables are present in registers.

The idea can be generalized for example to convert, for instance, `a==x||a==y` into `(a^x)|(a^y)`. We have implemented special function attempting to convert conditional into expression that is non-`NULL` when conditional is true, and use it to combine two instructions, algebraically simplify the resulting expressions and test whether resulting sequence is probably cheaper than the former. If so, we do the replacement.

We measured this optimization to improve SPEC2000 performance by about 0.3%. We expect the transformation to be more successful on EPIC architectures, but we have not tried it yet.

Chapter 9

Variable tracking

9.1 Introduction

Variable tracking is not an optimization pass. It computes information that helps the debug output to be better. It tracks where the variables are stored at each position in instruction stream. It is run as one of the last passes in GCC so that no pass could make this information out of date. The debug info is then generated according to this information.

Intel wrote the pass before us and it worked. But they used the code for writing of dwarf so that it could not be contributed to GCC. Cygnus tried to write similar thing with Live-Range Splitting (splitting a variable into several ones so that the variables that are not used in a loop can be moved to stack) and Stabs. Their solution was done by notes generated before Reload and nonstandard extensions and never came to GCC mainline.

People working on GDB like our solution and they are working on the support for it. But it is not done yet so the only way how to use Variable Tracking is use Intel compiler for i960 and run GDB for i960 modified by Intel in a simulator.

After our patch will be contributed to mainline, it will be possible to add the Live-Range Splitting and Webizer, the Register Renaming will be allowed to run with -O2[1] instead of -O3[1]etc. These optimizations will make the code generated by GCC even faster.

9.2 Implementation

First we have to know what is stored in a register (in RTL representation). We have added one more field to the RTL representation. This field contains a pointer to structure describing (a part of) a variable stored in register. We use a hash table for these structures. Information about content of a register is updated when using sub-registers, allocating registers or when doing Live-Range Splitting. There is also similar description of variable stored in memory.

The Variable Tracking pass does data-flow analysis. Computing data-flow on RTL itself would be complicated, so it first scans each basic block and remembers

[1]The level of optimization. Higher number means more optimization.

each use of register/memory in the basic block. Each such a record contains the register/memory, the instruction that it is in and the purpose (`LT_SET_DEST`[2], `LT_PARAM`[3], `LT_CLOBBERED`[4]) or the register/memory in the instruction.

Now we perform the data-flow analysis to propagate the variable locations. We use Hybrid Search Algorithm[8] that is faster than the traditional iterative or worklist algorithms (actually hybrid search algorithm is a combination of iterative and worklist algorithm). First, we initialize the IN and OUT sets for each basic block. The sets are actually arrays of link-lists, each link-list contains the information about (a part of) a variable that is stored in corresponding register. The memory references will be remembered in a hash table. We use link-list for registers because there can be more than one variable assigned to a register (see an example of code where register allocator can assign two variables to one register) but the number will be quite low so the link-list is probably the fastest solution.

Example: Register allocator can assign the same register to variable A and B in this code.

if condition **then**
  define value of variable A
else
  define value of variable B
end if
code that does not use A nor B
if condition **then**
  use A
else
  use B
end if

Then we clear the "visited" bitmap and set the "pending" bitmap and put all blocks to "worklist" (Fibonacci heap). The blocks in worklist are ordered in reverse completion order of Depth-First Search (DFS) which should speed up the data-flow analysis. We take the first block from the worklist and start search from it. Given a basic block, the IN set of the block is computed as a union of predecessors' OUT sets. Then we clear the pending bit and set the visited bit, and compute the OUT set from the IN set and the recorded purposes of each register/memory in the basic block. If the OUT set has changed we set the successors' pending bits. The search continues in the unvisited successors.

We are passing the locations of variables by `NOTE_INSN_VAR_LOCATION` notes to the debug info writer. Each note describes the location of one variable at the point in instruction stream where the note is. The note contains a list of pairs (offset, location) because variable can have several parts and each part has its own location (for example 64-bit integer variable on 32-bit machine has usually 2 parts). There is no need to emit a note for each variable before each instruction, we only emit these notes where the location of variable changes.

After the data-flow finishes, we know where the variables are in the beginning and the end of each basic block. We emit two groups of notes for each basic block:

[2] Reg/Mem is a destination of an assignment.
[3] Reg/Mem is a parameter of the instruction, it is used but not changed.
[4] Storing an unpredictable value to reg/mem.

1. Notes for the changes between the OUT set of the previous block and the IN set of the current block (the OUT set of the "previous" block of the first block is empty by definition) will be emitted **before** the head of actual basic block.

2. Notes for changes of variable locations because of the effects of instructions in actual basic block. If the purpose of the reg/mem in the instruction is `LT_SET_DEST` or `LT_CLOBBERED` the note will be emitted **after** the instruction that the reg/mem is in (because the instruction sets/destroys the value). If the purpose of the reg/mem is `LT_PARAM` the note will be emitted **before** the instruction because a variable is already in the location before this instruction.

9.3 Dwarf2

Dwarf2 is one of the first formats of debug info that supports the location lists of variables. The section of dwarf2 describing variables can contain location list. Basically it is the list of positions in code and for each position in code there is a list of changes of variables' positions. Daniel Berlin `<dan@dberlin.org>` was so kind and wrote the code for writing this debug information from `NOTE_INSN_VAR_LOCATION` notes for us.

Part II

Results

Chapter 10

Branch Predictors

10.1 Methodology

To verify validity of branch prediction heuristics (see chapter 6 for description of individual algorithms) we compile program with profile feedback, dump our predictions and real program behavior to the debug output and use our `analyze_branches` tool to produce summary.

We tested our results on SPECint2000 suite, a representative collection of programs run on nowadays computers. Since we used the same set of programs for specifying weights of individual heuristics, we decided to test SPECfp2000 suite to cross check our predictor on different code.

Programs contained in SPECfp differ dramatically from the ones in SPECint by both purpose and language. Many SPECfp programs solve numerical problems and are written in Fortran, while SPECint programs are written in C and C++.

We tested only subset of SPECfp tests, because some of them are written in f90 Fortran dialect that is not supported by GCC yet.

10.2 Results

We present the results in tables 10.1 and 10.2. The "algorithm" column specifies name of prediction heuristic as described in chapter 6. "first-match" heuristic refers to result of combining all strong prediction heuristics using first-match principle, while "DS theory" refers to result of combining the weak heuristics using Dempster Shaffer theory. Again see chapter 6 for details. "no prediction" refers to branches we failed to predict at all and finally "combined" refers to combination of all heuristics used by GCC.

The column, called "branches" specifies number of conditional jumps in all compiled programs predicted by given heuristic in both absolute and relative form.

One can see that SPECint contains 88411 conditionals and we predicted 15% of them using strong heuristics and 60% using weak heuristics. We failed to predict 24% of branches. SPECfp contains 20079 conditionals and we predicted 22% of them using the strong heuristics and 51% using weak heuristics. We failed to predict 26% of branches.

Algorithm	Branches	(rel)	Hitrate	Max	Coverage	(rel)
DS theory	53018	60.0%	70.61%	91.03%	11476302572	33.2%
call	26573	30.1%	70.48%	94.17%	5516113528	15.9%
combined	88411	100.0%	77.00%	91.82%	34591922166	100.0%
const return	637	0.7%	95.59%	98.81%	86727364	0.2%
continue	728	0.8%	56.00%	86.38%	1112290143	3.2%
early return	1006	1.1%	66.56%	77.62%	100847138	0.3%
first match	13529	15.3%	90.53%	94.06%	15070692296	43.6%
goto	1826	2.1%	70.57%	90.46%	28111884	0.1%
loop branch	6646	7.5%	89.44%	93.25%	7940058090	22.9%
loop exit	5437	6.1%	91.55%	95.49%	5846544398	16.9%
loop header	7581	8.6%	64.66%	90.14%	1771208607	5.1%
loop iterations	713	0.8%	91.44%	91.44%	1111820785	3.2%
negative return	272	0.3%	96.45%	97.43%	64283908	0.2%
`NULL` return	356	0.4%	90.52%	92.99%	58741977	0.2%
no prediction	21864	24.7%	60.35%	88.74%	8044927298	23.2%
noreturn call	976	1.1%	99.99%	99.99%	207850824	0.6%
values nonequal	23039	26.1%	69.88%	89.38%	4076339804	11.8%
values positive	2440	2.8%	78.58%	84.57%	1179093128	3.4%
pointer	6037	6.8%	82.83%	94.25%	968014026	2.8%

Table 10.1: Experimental results on subset of SPECint2000 benchmark suite.

Algorithm	Branches	(rel)	Hitrate	Max	Coverage	(rel)
DS theory	10261	51.1%	70.09%	96.69%	3972049376	18.0%
call	4775	23.8%	33.04%	95.45%	787258096	3.6%
combined	20079	100.0%	81.82%	91.76%	22061567792	100.0%
const return	40	0.2%	9.22%	90.77%	177957066	0.8%
continue	10	0.0%	1.58%	98.41%	7958	0.0%
early return	523	2.6%	88.38%	99.81%	7560387	0.0%
first match	4545	22.6%	90.44%	90.53%	14951251972	67.8%
goto	171	0.9%	99.93%	99.99%	5477020	0.0%
loop branch	2804	14.0%	90.60%	90.69%	12182517835	55.2%
loop exit	2151	10.7%	96.07%	96.40%	1945083921	8.8%
loop header	2710	13.5%	99.34%	99.43%	1190273758	5.4%
loop iterations	608	3.0%	74.65%	74.65%	833904864	3.8%
negative return	4	0.0%	100.00%	100.00%	4	0.0%
`NULL` return	69	0.3%	99.99%	99.99%	12579497	0.1%
no prediction	5273	26.3%	55.63%	91.41%	3138266444	14.2%
noreturn call	134	0.7%	99.99%	100.00%	1646663	0.0%
pointer	481	2.4%	97.89%	99.95%	68871540	0.3%
values nonequal	3298	16.4%	67.14%	94.90%	1813253392	8.2%
values positive	2165	10.8%	97.43%	98.37%	1283740708	5.8%

Table 10.2: Experimental results on subset of SPECfp2000 benchmark suite.

More important information than static instructions counts are the dynamic instruction counts—counts weighted by actual number of executions of each instruction in the train run. The values are presented in last column again in both absolute and relative form.

In SPECint we predicted 43.6% of executed instructions by strong heuristics and 33.2% by weak heuristics, that is in sharp contrast to static instruction counts. We failed to predict 23.2% of instructions. In SPECfp the numbers are even more biased in same direction—67.8% for strong heuristics, 18% for weak heuristics, and 14.2% unpredicted.

The hitrate column presents probably the most important value—the probability that conditional jump will go in the direction predicted by heuristics. We present only the dynamic numbers. For SPECint we predicted 77% of branches correctly and in SPECfp even almost 82%[1]. This compares well to 72% hitrate we measured by exact re-implementation of Ball and Larus [3] heuristics[2] that our work is based on and to roughly 80% hitrates reported by more expensive branch prediction methods based on intra-procedural weighted value range propagation [5] not yet cheap enough for production compilers.

Strong heuristics were very successful—90% in both cases, while DS theory reached about 70% success.

Last column represents values theoretically reachable by perfect static predictor that always predict branch in its more probable direction. So upper bound of hitrate for static branch prediction methods is about 91%, but it is unrealistic to expect this to be reached without profile feedback.

We would like also to point out that many of our new heuristics are very accurate—for instance the return value based ones. Also splitting single opcode heuristic to multiple ones helped us to increase the hitrate of "value positive" heuristic.

Finally figure 10.1 shows comparisons of hitrates of our combined heuristics and perfect predictor on each of SPECint2000 benchmarks. As can be easily seen, the sucesfulness of our method varies from program to program, but in all cases it has at least slightly successful.

We found that some of optimizers needs to be modified for properties of estimated profiles. For instance estimated profile almost always predict loop to have relatively few iterations making loop peeling too active, but overall almost all optimizations implemented for profile feedback derive some benefit from estimated profile too as shown in next chapter.

[1]Numerical programs generally have more predictable control flow.

[2]Ball and Larus tested their work on different set of programs resulting in about 80% static hitrate. Later tests on SPECint92 has shown 73% dynamic hitrate, result more comparable to ours

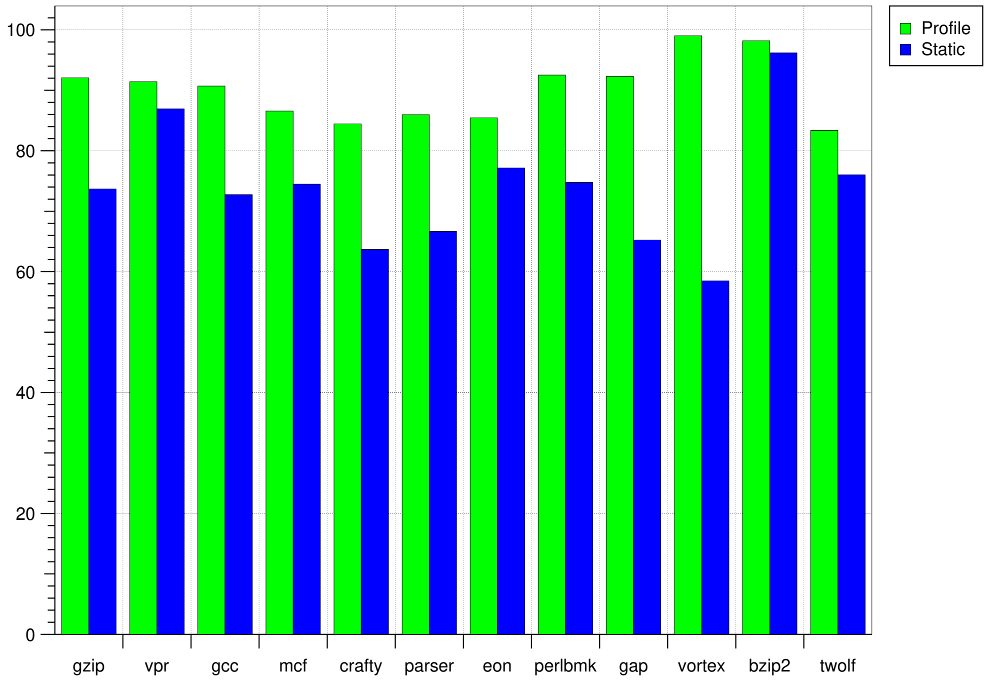


Figure 10.1: Comparison of perfect and static branch prediction on SPECint2000 tests.

Chapter 11

Performance measurements

In this chapter we present some benchmarks of our changes. It should be stressed that the optimization itself was not the major goal of our project that concentrated primarily on preparing GCC for easy integration of new optimizers, however of course the improvement of generated code was the main motivation and we implemented several new optimization passes, so we should test the benefits.

11.1 Methodology

Andreas Jaeger has kindly tested our code on industrial standard SPECint2000 benchmark suite. It contains several commonly used programs—for instance the perl interpreter, old version of GCC itself, chess playing program crafty, gzip and bzip compression programs and more. The detailed information about the product can be found at `http://www.spec.org`.

The SPECint2000 is highly memory bound as programs usually operate on data set of about 200MB in size and the code segments are also quite large making these results conservative estimation of our contribution.

Since we integrated our changes to mainline GCC tree containing changes from many other developers and also merged all changes happening in mainline to our development tree, there is no version of GCC without our changes comparable with our cfg-branch version available.

We decided to benchmark only the benefits of profile based optimizations, as these can be easily disabled. This is again just a conservative approximation of our work.

We also benchmarked GCC with two different sets of options. The `-O2 -march=athlon` commonly used by developers and with aggressive optimizations enabled `-O3 -fomit-frame-pointer -march=athlon -funroll-all-loops -fstrict-aliasing -malign-double -fprefetch-loop-arrays`. These flags were chosen to test as many features of GCC as possible, not to produce best performing code, and they are used by Andreas' automated tester to monitor GCC performance for almost a year now.

11.2 Results

The relative results for each SPECint2000 benchmark are in figures 11.1 and 11.2. Each benchmark contains 4 values all relative to performance of mainline GCC with profile estimation disabled. 2 are for mainline tree and 2 for cfg-branch[1]. Each tree is benchmarked first without profile feedback (ie. with profile guessed by our branch prediction methods) referred as "static" and later with profile feedback referred as "profile'.[2]

Andreas used AMD Athlon 1.133GHz system with 496MB of memory and GNU/Linux operating system for benchmarking.

As you can be easily see, with profile feedback both mainline and CFG-branch perform consistently better than without estimation bringing up to 14% speedups. In the geometric mean of all benchmarks, profile feedback improves performance by about 3% in the mainline and by about 4% in the cfg-branch.

The static profile estimators are successful enough to produce better code at average than with no predictions, but for crafty, parser and twolf we loose. By studying the `analyze_branches` outputs for these benchmarks we found that in the first two cases the hitrate is only about 63%. Crafty contains common loops that iterate exactly once, in the parser majority of pointers are NULL. Both these cases correspond to a single function in the source program. We may try to teach developers to use `builtin_expect` feature to help compiler in such case to get good performance.

Twolf is special. Hitrate of our branch predictor is very good—78% so we may want to investigate what really is going wrong. This may be just random effect of code layout changes, such as cache coloring conflict. On modern architectures, like Athlon is, it is very difficult to get consistent results in all cases because of number of factors affecting the final performance.

We shall also mention that cfg-branch aggressive optimization results already contain our new loop optimizer replacement. This makes the results less comparable to mainline. The feature set of our new implementation is much smaller than the old unroller, but as can be seen from results, it works better in majority of cases than the old code did. We started to work on the new unroller because we saw that the old code had to be replaced, but we did not even expected our unroller to outperform old code so soon.

Without estimation we measured 386.00 SPECints[3], with estimation 391.82 SPECints (1.3% improvement), with profile feedback 402.08 SPECints (4.1% improvement). Majority of benefits can be already measured in the mainline tree, as most of our new optimizations are not enabled.

In the aggressive optimization we measured 399.22 SPECints without estimation, 411.75 with estimation (3.1% improvement) and 415.92 with profile feedback (4.1% improvement).

The 1.3% improvement for static profile estimation with standard settings may look small, but we shall mention, that it already is more than benefits

[1] We present results from both trees, as the mainline changes will be released in next version of GCC, scheduled at April 15th, so users will benefit from them soon. We hope to integrate majority of changes on cfg-branch to the official tree once next release will be out, but we can't guarantee it.

[2] SPECint rules dictates that different data set is used for train run than later for benchmark run, so results should be realistic.

[3] SPECints are computed as geometric average of speedups all benchmarks relative to original SPEC test machine in percents. The higher, the better.

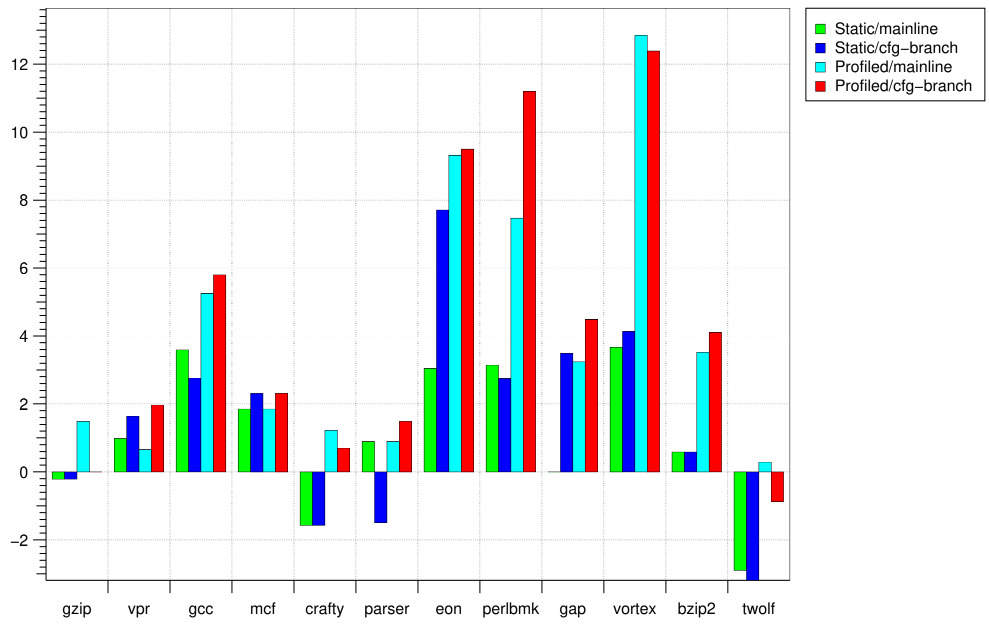


Figure 11.1: SPECint2000 comparison with standard flags.

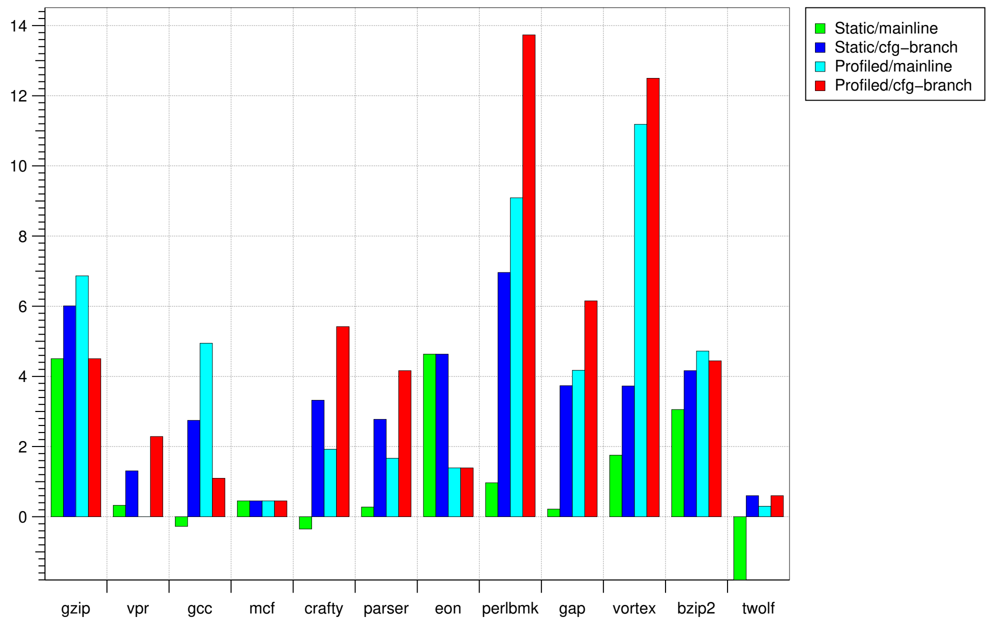


Figure 11.2: SPECint2000 comparison with aggressive optimization.

derived from instruction scheduling or loop optimization pass that have many times higher complexity and (in the first case) compilation time requirements.

Benefit of 4.1% compares realistically to improvements for profile feedback reported by other compiler teams. DEC Alpha compiler team reported results of their longer and more involved approach bringing 17% speedup on Spec95 testsuite. Majority of benefits (10%) came from function inlining heuristics we could not implement yet, since profile can not be gathered at Abstract Syntax Tree level. Once work done on ast-branch is integrated we plan to focus on it, but we can not implement it at the moment without duplicating the work being done by other developers.

Major benefits reported by DEC team came from tracer 3% and code reordering pass 4%, that we implemented too, but our measured benefits are lower (0.3% and 1.7%) even when we implemented more sophisticated algorithms. This can be explained by architectural differences between Alpha and Athlon chips. Athlon is designed to execute code optimized for different architecture of Intel's chips and has a lot of built in optimization logic making it much less responsible for (and dependent on) compiler optimizations. It has also on-chip scheduling which eliminates major benefits of tracer and the i386 architecture has much more compact code which partly eliminates the benefits of code layout algorithms. These purposes makes us believe that benefits will be much higher on other platforms. For instance profile estimation is a kind of requirement for proper compilation for modern EPIC architectures making our contribution more appealing in the near future. Unfortunately we can't run benchmarks on such machines at the moment.

To illustrate how nontrivial is to reach speedup in optimizing compiler we shall mention that according to results of Andreas tester, GCC 3.0.0 improved over GCC 2.95 by only 3% after two years long intensive effort. GCC 3.1.x will be likely about 6% faster and we believe important portion of that is due to our changes. The difference between `-O1` and `-O3` optimization level is only 6.4%[4], as a collaborative result of 25 new optimizations enabled at that level.[5]

In this light we see our results as success.

[4]See `http://www.suse.de/~aj/SPEC/`

[5]`-O1` enables just minimum of optimizations to keep compilation time short—jump optimizations, common subexpression elimination, dead code removal, jump threading and cross jumping, invariant code hoisting and improved register allocation.

Chapter 12

Summary

We believe that in our two semesters long effort we accomplished our original goal and exceeded it greatly, as our infrastructure improvements allowed us to implement several successful optimization passes (and few not so successful, see discussion in 11 and 12.1) in shorter time than before. Overall benefits we benchmarked are competitive with results reported by other compiler teams.

While the original reorganization of compiler to use CFG was mainly done by Jan Hubička in the early stages of our projects, the other participants of the project (Zdeněk Dvořák, Josef Zlomek and Pavel Nejedlý) were able to familiarize themselves with the necessary parts of the compiler in less than one month and use the CFG infrastructure to develop new optimization passes, improve importantly the GCC ability to predict the profile statically and bring framework to measure efficiency of individual branch prediction heuristics, and increase usability and robustness of basic block profiler implementation and when needed find and fix the latent GCC implementation problems. They also brought important feedback for design of CFG code, its documentation, and implemented several extensions (such as new natural loop tree representation, accurate debugging output support routines or robust profiler code).

A number of optimizations were implemented and we believe that mostly successful ones — among the most important ones, the register allocation changes, code placement and new loop unrolling code. All the new passes are significantly shorter and easier to maintain than older GCC code.

For instance the CFG code is shared across all optimizations. Code layout and basic block duplication module is implemented on using it and is reused in several optimization passes (block reordering, loop optimizer, tracer). We implemented natural loop discovery code in 1100 lines and loop body duplication code in 900 lines used by loop peeling, unrolling and unswitching. Our loop unroller is just 700 lines long as opposed to 4000 lines of code of monolithic unrolling code present in old loop optimizer that magically used 3000 lines of code for instruction chain duplication from function inlining module. Our new loop optimizer, even when it is feature wise much poorer than the old code, already outperforms the old code in SPEC2000 benchmark at much lower code size expanses and we got loop peeling and unswitching de-facto for free.

We have found our implementations to be significantly shorter than the ones present in Open64 and competitive with Impact compiler[1], the other two large compiler projects with sources publically available. We also believe our implementations are easier to understand, but we probably need to keep the judgment on independent reader.

We have successfully merged major infrastructural changes to GCC mainline tree. Important amount of work is already present in the 3.1 branch to be released in 5 days (April 15th). 3.1 version will be the first official GCC version supporting profile feedback and doing limited amount of profile based optimizations — register allocation, code alignment and simplistic basic block reordering.

Some more changes have been merged to the development tree, after it had been unfreezed last month, and only some of new optimizations are waiting to be integrated. Merging of the other changes is scheduled after the 3.2.0 release, because GCC maintainers are now focusing on pushing out the release.

Our special thanks come to Richard Henderson, one of the most active GCC maintainers, who kindly reviewed majority of our patches and approved them for inclusion into mainline and provided useful comments and ideas and even developed and improved our code in areas we were unsure about, and Andreas Jaeger, who tested and benchmarked our work using SPEC2000 benchmark suite.

12.1 Future Plans

While officially our project is about to be finished, we plan to continue working on the GCC. Of course we plan to maintain the changes we made, attempt to merge rest of them to the mainline GCC tree for next major release (3.2) and fix problems as they arise (we expect number of them on more exotic embedded platforms we can't test directly).

As shown earlier, our results can be seen as a satisfactory, but some optimizations does not perform as well as we hoped for. Most important is tracer, that brings much smaller speedups (0.3%) than reported by DEC Alpha compiler team (3%). Partly this can be attributed to differences between Athlon and Alpha architectures as discussed already, but partly it can be due to architecture of GCC itself. Many optimization passes are basic block based and thus do not benefit from extended superblock size. We want to update as many of optimizations as possible to be either superblock based or global.

We also plan to test the superblock scheduler, as superblock scheduler is one of the passes deriving most benefits from trace formation. We have not integrated a working prototype to the source tree yet and we want to see whether some speedups can be gained from it. Impact compiler is using exactly such scheme with great success on Itanium architecture. We will attempt to merge superblock scheduler to the mainline in case that the DFA-branch targeting more powerful global scheduler will not be able to deliver it for next release.

On AMD Athlon GCC current scheduler implementation brings about 0.9% speedup. This is much lower compared to other chips, because Athlon has an on-chip scheduler that is mostly replacing the compiler pass. We expect that

[1] An experimental compiler where many algorithms are very time consuming to keep source base simple.

more global scheduling can bring more benefits, as the on-chip scheduler is not able to perform it.

Important extensions are also desirable to our loop optimizer pass. We should re-implement the strength reduction pass and add more sophisticated unrolling heuristics and induction variable discovery. We also plan to add dependency analysis and array prefetch code generation. This work needs to be closely synchronized with effort on AST-branch, as loop optimizing framework should be distributed between the high level optimizers that are available there, and the lowlevel RTL optimizers we are working on. For instance we do not plan to re-implement the induction variable discovery code in full strength as it is present in old loop optimizer, instead use highlevel optimizer to canonicalize loops to make our job easier. At high level some important information is present (such as whether the counter overflows) we can't derive from RTL representation, so low level optimizers cannot do as good job as high level ones in this case.

Important step will be extending our profile implementation to the tree representation, that is necessary to implement function inlining heuristics without requiring user to profile program twice. Profile based inlining has been reported as the most successful optimization among profile based changes made by the DEC Alpha compiler team. Again we need to wait for AST-branch to mature enough to make this task possible.

Similarly once inter-procedural framework is available, we plan to extend our profile estimation to the inter-procedural level allowing us to predict which functions belong to the hot spots of program and which do not. This has been reported as reliable in Wu and Larus paper[4].

Lots of work need to be done on midlevel RTL representation. While our prototype works reliably on i386 machine, other machine descriptions need to be updated to handle it. We also need to implement some side corners, such as string representation and function calls, we do not represent in midlevel RTL yet. We plan to continue working on making midlevel RTL more high to simplify the RTL generation pass and add multiple passes gradually lowering similarly to the SGI MipsPro compiler design. For instance we would like to have array subscripts represented directly in the midlevel RTL in early stages to simplify dependency analysis.

Interesting direction of future research can be trying to implement path profiling to the GCC. Path profiling is scheme able to measure whether outcome of given branch depends on the path it has been reached from. In case it does, the code can be duplicated (specialized) to assist hardware branch predictor and improve optimizations. Path schedules are also easier to update after other code specializing optimizations, such as loop unrolling and loop peeling. Similarly we plan to add code to measure histograms of number of iterations of each loop so we can better decide what loops to peel or unroll and how much.

We also plan to replace GCC register move pass by extending our register coalescing and implement more optimizers. Our existing optimizers will also definitely benefit from some extra tunning and benchmarking especially on non-i386 architectures.

Important cleanups to the control flow graph representation are possible as well. For instance once all passes are updated to use control flow graph information, we can unlink independent basic blocks in the instruction chain. We should also avoid the notes and jump tables to be present between the basic blocks. This requires high volume changes, so we plan to synchronize this with mainline development to avoid the need to maintain large patch for a too long time.

Part III

Project Documentation

Chapter 13

Organization

13.1 Credits

The project was leaded by David Bednárek[1]. Following students participated in the project directly:

Zdeněk Dvořák [2]

Zdeněk implemented the thread safe profiling code, added infrastructure for high level branch predictor algorithms.

Major part of his work is related to loop optimizations. He implemented new data structure to hold natural loop tree and modified natural loop discovery code originally contributed by Michael Hayes to produce it. On the top of it he implemented loop unswitching, loop unrolling and loop peeling algorithms, one of the most successful new optimizations in our project.

He also made numerous contributions to the control flow graph handling code, most importantly he changed the basic block representation from linearly ordered array to double linked list avoiding quadratical complexity of many important algorithms in new CFG aware code. This required many changes of code in compiler that uses basic blocks.

He also found and fixed many latent bugs in the compiler and our new improved CFG manipulation code.

Jan Hubička [3]

Jan had special role in project by being the only participant with previous knowledge of GCC internals and its development model. In early stages of project he helped other participants to familiarize with project by solving relatively simple problems and has been concentrating on adding infrastructure necessary for CFG and profile manipulation. He also added new branch prediction heuristics and a pass to estimate profile.

Predating the official start of project, he implemented first prototype of working basic block profiler, extended branch prediction pass and modified

---

[1] `bednarek@ulita.ms.mff.cuni.cz`

[2] `rakdver@atrey.karlin.mff.cuni.cz`

[3] `hubicka@ucw.cz`

few passes to use the information. Since the information was not maintained only register allocation used the information. He also implemented bare bones of future cfg-cleanup pass.

In later stages of project he implemented few new optimization passes (tracer, webizer, register coalescing), but continued to concentrate on GCC cleanups and infrastructural changes and later on mid-level RTL implementation. He also reviewed majority of contributions of other participants before integrating to the source tree.

Pavel Nejedlý [4]

Major part of Pavel's work is concentrated on the basic block profiler. He implemented new, robust and extensible file format to hold profiles. His format allows overall information to be summarized which is used to identify better hot spots in the program and to do code placement decisions as well as user errors (such as using outdated profiles) to be detected and reported. In future his work will allow more sophisticated profile merging algorithm, as new format stores each train run separately instead of merging them all at once.

Pavel has also reimplemented the interface between profile instrumentation generated by compiler and the libgcc runtime in cleaner and more extensible way.

Last his major contribution is new data structure to hold dominance information that allows fast dynamical updates and is necessary for Zdeněk's new loop optimizer code.

Josef Zlomek [5]

In early stages Josef has found and fixed major problem of Michael's natural loop discovery code. Later he has implemented and tuned the new basic block reordering algorithm based on software trace cache. His implementation is in many way superior to the one described by original article. He also improved and fixed the cfglayout module he used for his work.

Most important contribution is probably the new variable tracking pass fixing the aged defect of GCC debugging information output code allowing us to enable more optimizations at default level.

Josef has also reviewed lots of code done by Jan Hubička and provided useful comments and fixed important bugs.

Following people has contributed to our project:

Andreas Jaeger

Andreas set up the SPEC2000 tester to do daily benchmarking of our development tree which allowed us to easily tune new optimizations on nontrivial benchmark suite. He also set up an extended weekly tester which provided feedback for our branch prediction heuristics, and kindly benchmarked results of our work that are presented in this documentation.

[4] `bim@atrey.karlin.mff.cuni.cz`
[5] `zlomek@matfyz.cz`

Richard Henderson
Richard, one of the most active maintainers of GCC, has reviewed and approved for inclusion to the mainline the essential parts of our work. He also provided number of important ideas. Without his help our project would not be possible at all. He also helped us to redesign scope tree maintenance code in `cfglayout.c`.

Jeff Law
Jeff pushed us to implement midRTL.

Daniel Berlin
Daniel implemented and tested Dwarf2 output routine based on Josef's variable tracking pass. He also tested and fixed our version of compiler on powerPC and added improvements to the loop unrolling debug output code.

13.2 Documentation

Our documentation has been written by all participants of the project each describing his portion of work. The printed section contains introduction part and experimental results. The later section describing data structures and interfaces has been committed for inclusion in the official GCC manual [1].

Major portion of documentation however is in the sources. This is dictated by GNU Coding Standards [1] and maintainers of GCC project. Since GCC contains a lots of aged and complicated code, good documentation is essential.

Each function should have description of purpose of its arguments and return value, in addition to commenting all nontrivial steps in the function body. Each of our newly added modules contains summary of algorithm in the introductory comment.

13.3 Sources

Since our project is integrated to the GCC source tree, we provide short overview of locations of our changes. We have implemented from scratch following modules:

`cfgcleanup.c`, `cfgloop.c`, `cfgloopanal.c`, `tracer.c`, `web.c`, `coalesce.c`, `loop-new.c` `unroll-new.c`, `unroll-new.c` `var-tracking.c`, `bb-reorder.c`[6], `midrtl.c`, `et-forest.c`, `et-forest.h`, `predict.def` and `analyze_branches` script.

We made major changes to following modules:

`flow.c` We split out the CFG handling code into `cfg.c`, `cfganal.c`, `cfgrtl.c`, `cfgloop.c` and importantly changed (de facto reimplemented) all the contents. We also added support for easy dynamic updating of liveness information.

`emit-rtl.c` We modified all routines to update the flow graph.

`toplev.c` We reorganized the `rest_of_compilation` function.

[6] the contents of original bb-reorder was either replaced, or moved into `cfglayout.c`

`cfglayout.c` We rewrote the code to handle syntactic scope nest tree and changed all interfaces. We added functionality for code duplication and reorganized the way how the insn chain is put together.

`gcse.c` GCSE now maintains liveness information and uses our code hoisting infrastructure code to avoid limitations of previous implementation.

`df.c` Since we are the first users of Michael Hayes' data flow module, we had to fix bugs and some design defects. We also cleaned up the code.

`gen*.c` All these functions had to be updated to handle midRTL.

`recog.c` The most of mid level RTL code is present here.

`optabs.c` We had to modify all RTL generation routines to produce mid level RTL code.

`ifcvt.c` contains double test optimization pass.

`predict.c` contains new branch predictor implementation.
We basically rewrote this module from scratch.

`profile.c`, `final.c`, `libgcc.c` contain updated profiler implementation and changed interfaces.

`reg-stack.c`,`reg-class.c`,`local.c`,`global.c` Updated to use the profile.

`stmt.c` contains high level branch predictors.

Because of the nature of project, we had to touch almost all other back-end modules in GCC as well.

13.4 Timeline

Exact progress of the project is described in `ChangeLog`, `ChangeLog.6` and `ChangeLog.cfg` files in the GCC subdirectory, but here we point out some dates we consider especially important for our project.

Jul 28 First prototype of new branch predictor and profiler integrated to GCC + modification of register allocator. (Jan Hubička)

Sep 10 First version of cfgcleanup integrated. (Jan Hubička)

Sep 25 Large reorganization of CFG related modules. `flow.c` breakup (Jan Hubička)

Nov 12 CFG efforts moved from mainline to cfg-branch due to destabilization.

Nov 12 Natural loop discovery code finally works. (Josef Zlomek)

Nov 12 Cross-jumping and jump threading merged to mainline. (Jan Hubička)

Nov 13 Old jump optimization pass is finally dead. (Jan Hubička)

Nov 15 High level branch prediction integrated. (Zdeněk Dvořák)

Nov 17 CFG layout duplication code and preliminary tracer implementation. (Jan Hubička)

Nov 26 Double test combining pass. (Jan Hubička)

Dec 11 First version of software trace cache implementation. (Josef Zlomek)

Dec 12 Tracer now produces better code.

Dec 13 Major of our changes are now in the mainline tree. Thanks to Richard Henderson.

Dec 15 GCC mainline feature freeze.

Jan 1 New profiler file format. (Pavel Nejedlý)

Jan 2 Thread safe profiling integrated. (Zdeněk Dvořák)

Jan 9 Software trace cache now produces better code (new loop rotation code). (Josef Zlomek)

Jan 16 Code hoisting infrastructure and GCSE revamp. (Jan Hubička)

Jan 21 Midlevel RTL prototype. (Jan Hubička)

Jan 23 Variable tracking code. (Jan Hubička)

Jan 24 New natural loop code. (Zdeněk Dvořák)

Feb 13 Finished integration of most of cleanups, fixes and infrastructural updates to the mainline. Thanks to Richard Henderson.

Feb 15 Final mainline freeze.

Feb 19 Major `cfglayout` cleanups and fixes. (Jan Hubička)

Feb 21 Loop code updates integrated. (Zdeněk Dvořák)

Feb 25 Loop unswitching code. (Zdeněk Dvořák)

Mar 18 Loop unrolling and peeling code. (Zdeněk Dvořák)

Mar 25 New debug output code. (Daniel Berlin, Josef Zlomek)

Apr 1 Feature freeze.

Apr 3 PowerPC and Sparc bootstrapped.

Apr 9 Fast dominance tree updating code integrated. (Pavel Nejedlý)

Chapter 14

Control Flow Graph

A control flow graph is a data structure built on top of the intermediate code representation (RTL instruction chain or trees) abstracting the control flow behavior of compiled function. It is an oriented graph where nodes are basic blocks and edges represent possible control flows from one basic block to another.

14.1 Basic Blocks

The basic block (node of control flow graph) is defined as a structure:

```
typedef struct basic_block_def {

  /* The first and last insns of the block.  */
  rtx head, end;

  /* The first and last trees of the block.  */
  tree head_tree;
  tree end_tree;

  /* The edges into and out of the block.  */
  edge pred, succ;

  /* Liveness info.  */

  /* The registers that are modified within
     this block.  */
  regset local_set;
  /* The registers that are conditionally
     modified within this block.
     In other words, registers that are set only
     as part of a COND_EXEC.  */

  regset cond_local_set;
  /* The registers that are live on entry to this block.  */
  regset global_live_at_start;
  /* The registers that are live on exit from this block.  */
```

```
  regset global_live_at_end;

  /* Auxiliary info specific to a pass.  */
  void *aux;

  /* The index of this block.  */
  int index;

  /* Loop info.  */

  /* The loop depth of this block.  */
  int loop_depth;

  /* Outermost loop containing the block.  */
  struct loop *loop_father;

  /* Immediate dominator.  */
  struct basic_block_def *dominator;

  /* Expected number of executions: calculated in profile.c.  */
  gcov_type count;

  /* Expected frequency.
     Normalized to be in range 0 to BB_FREQ_MAX.  */
  int frequency;

  /* Various flags.  See BB_* below.  */
  int flags;
}
```

Basic block is a straight-line sequence of code that can be entered only at the beginning and leaved only at the end. Basic blocks are represented using `basic_block` data type that contains pointers to the head (first instruction of the basic block) and the end (last instruction) as well as other information maintained about each block.

In the RTL function representation, the head pointer always points either to `NOTE_INSN_BASIC_BLOCK` or to `CODE_LABEL`, if present. Basic block ends by control flow instruction or last instruction before following `CODE_LABEL`.

Special basic blocks `ENTRY_BLOCK_PTR` and `EXIT_BLOCK_PTR` represent possible entry points and exits from the compiled function.

The `BASIC_BLOCK` array contains all basic blocks in the order they appear in the insn stream.

The RTL instruction stream contains not only the "real" instructions, but also notes. Notes may or may not appear inside basic block. Any function that moves or duplicates the basic blocks needs to take care of updating of these notes. Many of notes expect that code consists of linear regions making such updates difficult.

Additionally the jump table vectors are represented as "instructions" inside the insn chain. These vectors never appear in the basic block and should be always placed just after table jump instructions referencing them. After removing the table-jump it is difficult to eliminate the code computing address and referencing the vector, so cleaning up the vectors is postponed to liveness analysis and thus the vectors may appear in the insn chain without any purpose. Before any edge is made fall-thru, the existence of such construct in the way needs to be checked by calling `can_fallthru` function.

14.2 Edges

Edges in the control flow graphs are described by structure:

```
/* Control flow edge information.  */
typedef struct edge_def {
  /* Links through the predecessor and successor lists.  */
  struct edge_def *pred_next, *succ_next;

  /* The two blocks at the ends of the edge.  */
  struct basic_block_def *src, *dest;

  /* Instructions queued on the edge.  */
  rtx insns;

  /* Auxiliary info specific to a pass.  */
  void *aux;

  int flags; /* see EDGE_* below  */
  int probability; /* biased by REG_BR_PROB_BASE */
  gcov_type count; /* Expected number of executions calculated
   in profile.c  */
} *edge;
```

Edges represent possible control flow transfers from the end of basic block to the head of another. Single linked lists of edges to predecessors and successors are maintained for each basic block.

In common case edges correspond to branches or "fall-thru" edges to the following basic block, but there are several other reasons why edge may be created. The type of edge can be obtained via `flags` field.

There are following types:

jumps Edges corresponding to destinations of jump instructions have no flags defined. These edges are used for unconditional or conditional jumps and table jumps and are most convenient to manipulate with as they may be freely redirected.

fall-thru Fall-thru edges are present in case the basic block may continue execution to the following one without branching and do have `EDGE_FALLTHRU` flag set.

Unlike other types of edges, these edges must come into following basic block in the insn stream and thus function `force_nonfallthru` is available to insert jump in the case that redirection is needed. This may require creation of a new basic block.

exception handling Exception handling edges represent possible control transfers from trap to the exception handler. The definition of trap varies. In C++ only function calls can throw, but for Java exceptions like division by zero or segmentation fault are defined and thus each instruction possibly throwing this kind of exception needs to be handled as control flow instruction.

The destination of edge is specified by `REG_EH_REGION` note attached to the insn.

The flags of such notes set `EDGE_EH` and `EDGE_ABNORMAL`. In case of the call `EDGE_ABNORMAL_CALL` flag is set too.

When updating the insn stream it is easy to change possibly trapping instruction to non-trapping. Opposite conversion is difficult and should not happen. Predicate `may_trap_p` may be used to check whether instruction still may trap or not. The edges can be eliminated via `purge_dead_edges` call.

sibling calls Sibling calls terminate the function in a non-standard way and thus an edge to the exit must be present. `EDGE_ABNORMAL_CALL` and `EDGE_ABNORMAL` are set in such case.

computed jumps Computed jumps contain edges to all labels in the function referenced from the code. All those edges have `EDGE_ABNORMAL` flag set. The edges used to represent computed jumps often cause compile time performance problems, since functions consisting of many taken labels and many computed jumps may have very dense flow graphs, so these edges need to be handled with special care.

nonlocal goto handlers GCC allows nested functions to return into caller using `goto` statement referring to label passed to as an argument. The labels passed to nested functions contain special code to cleanup after function call. Such section of code is referred as *nonlocal goto receivers*. In the case function contains such nonlocal goto receivers, the edge from the call to label is present having `EDGE_ABNORMAL` and `EDGE_ABNORMAL_CALL` flags set.

function entry points By definition, execution of function starts by basic block 0, so there is always an edge from entry block to the first real basic block. The alternate entry points are specified by `CODE_LABEL` with `LABEL_ALTERNATE_NAME` defined. This feature is currently used for multiple entry point prologues and is limited to post-reload passes only. In future full support for multiple entry functions defined by Fortran 90 needs to be implemented.

function exits In the pre-reload representation function terminates by the last instruction in the insn chain and no explicit return instructions are used. This corresponds to the fall-thru edge into exit block. After reload optimal RTL epilogues are used, that use explicit (conditional) return instructions that are represented by edges with no flags set.

14.3 The Profile

In many cases compiler must make choice whether to trade speed in one part of code for speed in another, or trade code size for code speed. In such cases it is useful to know information about how often given block executes and that is the purpose for maintaining profile within flow graph.

GCC allows the profile to be either feedback based or statically estimated.

The feedback based profile is produced by compiling the program with instrumentation, executing it on the train run and reading the numbers of executions of basic block edges back to the compiler while re-compiling the program to produce final executable. This method provides very accurate information about where program spends most of time on the train run. Whether it matches the average run depends, of course, on the choice of train data set, but several studies has shown, that the behavior of program usually changes just marginally over different data sets.

When profile feedback is not available, compiler attempts to predict the behavior of each branch in the program using a set of heuristics (see `predict.def` for details) and compute estimated frequencies of each basic block by propagating the probabilities over the graph.

Each basic block contains two fields — `frequency` and `count`. Frequency is an estimation how often is basic block executed within a function and is represented as integer scaled in the range 0–`BB_FREQ_BASE`. Most frequent basic block in function is initially set to `BB_FREQ_BASE` and rest of frequencies are scaled according to that. During optimization, the frequency of most frequent basic block can both decrease (for instance by loop unrolling) or grow (for instance by cross-jumping optimization).

The `count` contains number of execution measured during training run and is nonzero only when profile feedback is available. This value is represented as 64bit integer. Most optimization passes can use only the frequencies of basic block, while few passes may want to know exact counts. The frequencies should always match the counts after scaling, however during updating of the profile information numerical error may accumulate into quite large errors.

Similarly each edge contains probability field—an integer in range from 0 to `REG_BR_PROB_BASE`. It represents probability of passing control from the end of source basic block to the destination. The probability that control flow arrives via given edge to the destination basic block is called reverse probability and is not directly represented, but it may be easily computed from frequencies of basic blocks. `EDGE_FREQUENCY` macro is available to compute how frequently is given edge taken. `count` field is present for each edge as well, representing same information as for basic block.

The basic block frequencies are not represented in the RTL instruction stream, the edge frequencies are represented only for conditional jump via `REG_BR_PROB`, since they are used when instructions are output to the assembly file and flow graph is no longer maintained.

14.4 Structure of profile output file

The name of the file corresponding to a source file is formed simply by changing file suffix to `.da`, e.g. for `test.c` the corresponding file is `test.da`.

As mentioned before, the file consists of several sections, each represening single run. The structure of each section is shown on figure 14.1

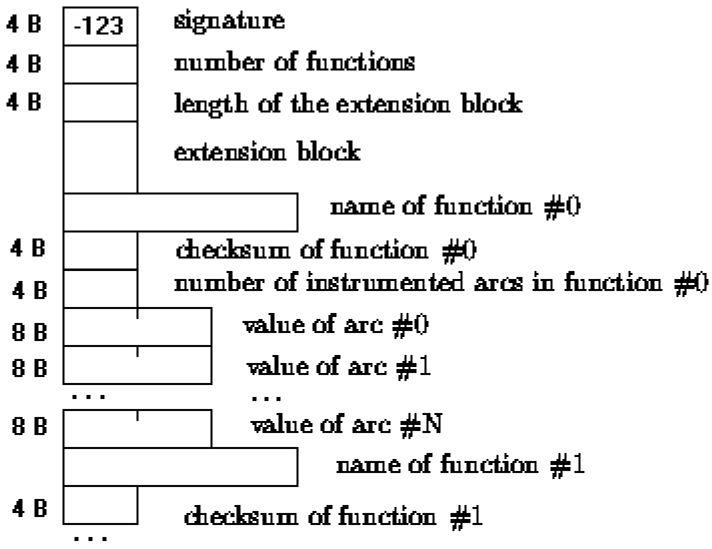


Figure 14.1: Structure of a section in profile data file

The function name in the figure is stored as a -1 (4 bytes), the length (4 bytes), the name itself (padded to 4-byte boundary) followed by a -1 (4 bytes).

The extension block is used for storage of other important data which may be emited by future versions of gcc. This allows use of particular profile with different versions of gcc.

In current version, extension block contains the following information:

1. number of instrumented arcs in whole program (4-byte number)

2. sum all of instrumented arcs in whole program (8-byte number)

3. maximal value of counter in whole program (8-byte number)

4. number of instrumented arcs in the object file (4-byte number)

5. sum all of instrumented arcs in the object file (8-byte number)

6. maximal value of counter in the object file (8-byte number)

The content of the file can be examined by utility `gcov`, which outputs the corresponding source file together with execution counts for each line (so-called line coverage). There is currently no utility for manipulating the profile output file structure, e.g. removing runs or merging two files together.

14.5 Maintaining CFG up to Date

Important task is to keep both control flow graph and profile up-to-date with the instruction stream during optimization passes. Reconstruction of control flow graph after each pass is not an option, as it is too expensive and we lose profile information.

At the moment, the basic block boundaries are maintained transparently during emitting instruction, so rarely there is need to move them manually (such as in case someone wants to output instruction outside basic block explicitly). Each instruction has defined `BLOCK_FOR_INSN` value that represents pointer to the basic block owning it, so the basic block list may be better viewed as integral part of instruction chain, than structure built on the top of it.

Updating of edges is not transparent and optimization pass is required to do that manually. However only few cases occur in practice. Commonly the optimization pass simplifies the instruction possibly eliminating some edge. This may happen by simplifying the conditional jump into unconditional, but also by simplifying possibly trapping instruction to non-trapping while compiling Java. The pass may call `purge_dead_edges` on given basic block to remove unneeded edges, if any.

Other common scenario is redirection of branch instructions, but this is best modeled as redirection of edges in the control flow graph and thus use of `redirect_edge_and_branch` is preferred over more low level functions, such as `redirect_jump` that operate on RTL chain only.

Last common case is inserting control flow instruction into middle of basic block. The `find_sub_basic_blocks` may be used to split existing basic block and add necessary edges, or `split_block` may be used when instruction in middle of basic block wants to become target of branch instruction.

For global optimizer, a common operation is to split edges in the flow graph and insert instruction to them. This can be easily done via function `commit_insn_to_edge` that emits instruction "to the edge" caching it for later `commit_edge_insertions` call that will care creation of new basic block where needed and flushing the instruction to actual instruction stream.

While debugging the optimization pass, an `verify_flow_info` function may be useful to find bugs in the control flow graph updating code.

14.6 Maintaining Profile up to Date

More delicate task than updating control flow graph is to update profile. Many of the function to modify flow graph, like `redirect_edge_and_branch` do not have enough information to easily update profile, so updating profile is in majority cases left on the caller. Since it is difficult to discover bugs in the profile updating code, as they manifest themselves only by producing worse code and checking profile consistency is not possible, because of numeric error accumulation, special care needs to be taken into this issue.

It is important to point out, that `REG_BR_PROB_BASE` and `BB_FREQ_BASE` are both set low enough to be possible to compute second power of any frequency or probability in the flow graph, it is not possible to even square the `count` field, as modern CPUs are fast enough to execute $2^3 2$ operations quickly.

14.7 Liveness Information

Liveness information is useful to determine whether register X is "live" at given point of program, that means that it contains important value. This information is used, for instance, during register allocation pass, as the pseudo registers need to be assigned to unique hard register or stack slot only when they are live. The hard registers and stack slots may be freely reused for other values when they are dead.

The liveness information is stored partly in the RTL instruction chain and partly in the flow graph. RTL chain stores local information: each instruction may contain `REG_DEAD` note representing that value of given register is no longer needed or `REG_UNUSED` note representing that the value computed by instruction is never used. The second is useful for instructions computing multiple values at once.

Each basic block contains bitmaps representing liveness of each register at entry and exit of basic block (`global_live_at_start` and `global_live_at_end`). `flow.c` contains function to compute liveness of each register at any given place in the instruction stream using this information.

Liveness is expensive to compute and thus it is desirable to keep it up to date during optimization passes. This can be easily accomplished using `flags` field of basic block. The functions modifying instruction stream automatically set `BB_DIRTY` flag of basic block, so the pass may simply use `clear_bb_for_blocks` before doing any modifications and then ask dataflow modulule via function `update_life_info_in_dirty_blocks` to get liveness updated.

This scheme works reliably as long as no control flow graph transformations are done. The task of updating liveness after control flow graph changes is more difficult as normal iterative data flow may produce invalid results or get into cycle when the initial solution is not bellow the desired one. Only simple transformations, like splitting basic blocks or emitting to the edge are safe, as functions to implement them already know how to update liveness locally.

14.8 Loop Tree

Loop tree describes the structure of loops. Nodes correspond to loops, while edges reflect the subloop/superloop stucture. Root of the tree is a dummy loop containing the whole function (including `ENTRY_BLOCK_PTR` as latch and `EXIT_BLOCK_PTR` as header).

Struct loop is defined as follows:

```
struct loop
{
  /* Index into loops array.  */
  int num;

  /* Basic block of loop header.  */
  basic_block header;

  /* Basic block of loop latch.  */
  basic_block latch;
```

```
  /* Number of blocks contained within the loop.  */
  int num_nodes;

  /* The loop nesting depth.  */
  int depth;

  /* Array of superloops of the loop.  */
  struct loop **pred;

  /* The outer (parent) loop or NULL if outermost loop.  */
  struct loop *outer;

  /* The first inner (child) loop or NULL if innermost loop.  */
  struct loop *inner;

  /* Link to the next (sibling) loop.  */
  struct loop *next;

  /* Loop that is copy of this loop
     (must be valid only just after copying the loop).  */
  struct loop *copy;

 ... /* Fields specific for the old loop optimizer
        are omitted.  */
};
```

The whole loop tree (plus additional information) is stored in struct loops:

```
struct loops
 {
  /* Number of natural loops in the function.  */
  int num;

  /* Array of pointers to loops.  Subloops should always
     have greater index than their parents.  */
  struct loop **parray;

  /* Pointer to root of loop hierarchy tree.  */
  struct loop *tree_root;

 ... /* Fields specific for old loop optimizer
        are omitted.  */
 };
```

Field `loop_father` of `basic_block` is a pointer to the innermost loop containing the basic block.

Functions for building, modifying and querying the loop structure are provided in `cfgloop.c` and `cfgloopanal.c`; some higher level functions for manipulating it may be found in `loop-new.c`.

Bibliography

[1] Free Software Foundation, *Using the GNU Compiler Collection (GCC)* `http://gcc.gnu.org/onlinedocs/gcc-3.0.4/gcc.html`

[2] R. Cohn, P. G. Lowney. *Design and Analysis of Profile-Based Optimization in Compaq's Compilation Tools for Alpha.* Journal of Instruction-Level Parallelism 3 (2000) 1-25

[3] T. Ball and J. Larus. *Branch Prediction for Free.* PLDI 1993.

[4] W. Wu and J. Larus. *Static Branch Frequency and Program Profile Analysis.* MICRO-27.

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

[6] C. Young, D. S. Johnson, D. R. Karger, M. D. Smith. *Near-optimal Intraprocedural Branch Alignment.* ACM 1997

[7] A. Ramirez, J. L. Larriba-Pey, C. Navarro, J. Torrellas and M. Valero. *Software Trace Cache.* International Conference on Supercomputing 1999

[8] D. C. Atkinson, W. G. Griswold. *Implementation Techniques for Efficient Data-Flow Analysis of Large Programs.*

[9] P. P. Chang, S. A. Mahlke, and W. W. Hwu. *Using Profile Information to Assist Classic Code Optimizations.*, ACM 1991

[10] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, B. Zorn *Evidence-based Static Branch Prediction using Machine Learning*, ACM 1996