# A Maintenance Programmer's View of GCC

Zachary Weinberg (CodeSourcery, LLC)
zack@codesourcery.com

May 3, 2003

## Abstract

GCC is considered more difficult to modify or debug than other programs of similar size. This paper will investigate the reasons for this difficulty, from the point of view of a maintenance programmer: someone producing a small patch to fix a bug or implement a feature, without causing new problems for unrelated use. Because the development tree's head is expected to be functional at all times, such incremental changes are normal—even regular contributors are in the maintenance programmer's shoes.

## 1 Introduction

Who is a maintenance programmer? Anyone working to implement a specific feature, or fix a specific bug, without introducing new problems at the same time. Anyone with limited time to investigate the situation and become familiar with the code.

Maintenance programmers are faced with both technical and procedural hurdles. GCC has a complex task to accomplish, but even so GCC is far more complicated than it needs to be, which makes it harder to modify the code than it should be. Further, once one does successfully make a change, it is hard to get it accepted to the official source tree. The procedural requirements are stringent for good reason, but still discourage people from contributing, and cause patches that were 90% correct to be rejected.

GCC's development process requires everyone to work incrementally and make minimally invasive changes. Although not a formal requirement, it is a consequence of the no-regressions policy for checkins, coupled with the extreme complexity of the source code. A simple change might turn out to have ramifications everywhere. A few individuals know the compiler inside out; they can pull off hugely invasive changes without breaking anything. Most of us are not that good, so we must take small steps, testing carefully as we go. Furthermore, even regular contributors often have difficulty getting their patches approved. And, of course, we all have lots of demands on our attention, so there is never enough time to work out the perfect design. Therefore, making life easier for a maintenance programmer who might have just one patch to contribute will make regular contributors' lives easier as well.

## 2 Technical Hurdles

Let's take a moment and look at the GCC source tree from 10,000 feet up. Table 1 breaks down the code by category. There are about 1.6 million lines in total, ignoring comments. Just over half of this is C; there are also substantial bodies of Ada, Java, and C++. Machine description files are written in a domain-specific, Lisp-like language, which accounts for ten percent of the total.

By nature, any program of this size is going to be nontrivial to work with. Furthermore, a compiler is necessarily more complicated than the average program of similar size, since it contains many algorithms and techniques that require arcane theoretical knowledge to understand. SSA (static single assignment) form, for instance, takes a good chapter of exposition to explain. GCC is necessarily more complicated than the average compiler, since it supports so many input languages and target architectures in its official distribution alone. Many other compilers support only one or two targets.

Even so, GCC's code could be much simpler and easier to maintain. This can be put down to three

| By category | | | By language | | |
|---|---|---|---|---|---|
| Core compiler | 250,000 | | C | 861,000 | 53% |
| Back ends | 410,000 | | Ada | 298,000 | 18% |
| biggest | 40,000 | (rs6000) | MD | 170,000 | 10% |
| smallest | 2,200 | (fr30) | Java | 127,000 | 8% |
| median | 6,500 | (v850) | C++ | 105,000 | 6% |
| Front ends | 480,000 | | Other | 78,000 | 5% |
| biggest | 221,000 | (ada) | | | |
| smallest | 2,500 | (treelang) | | | |
| median | 59,000 | (java) | | | |
| Runtime libraries | 458,000 | | | | |
| biggest | 274,000 | (java) | | | |
| smallest | 8,200 | (objc) | | | |
| median | 11,000 | (f77) | | | |
| Total | 1,639,000 | | | | |

Physical source line counts, generated using SLOCCount [1]. MD = machine description.

Table 1: GCC 3.3 source code breakdown

primary causes: incomplete transitions, functional duplication, and inadequate modularity.

### 2.1 Incomplete transitions

Incomplete transitions occur whenever anyone invents a new, better way to do something, but does not update every last bit of code that used to do it the old way. They might run out of time; they might not have the necessary expertise; they might just not be able to find it. Whatever the reason, an old API cannot be removed from the compiler until there are no remaining uses. An incomplete transition thus means that for an extended period there are two or more ways to do something. One is preferred, but it may not be obvious which. Someone writing new code that needs to do whatever it is, might pick the obsolete technique, further delaying the day when the old API can be removed.

Incomplete transitions are most common in the API for writing architecture back ends. For example, there are two ways to write a machine-specific peephole optimization. Both do pattern matching on the stream of RTL insns constituting the intermediate representation of a function. The old way (`define_peephole`) overrides the normal mechanism for writing out assembly language, substituting its own text. No further optimization can happen to the result. The new way (`define_peephole2`) replaces the matched insns with new ones, which can then be optimized further. For instance, the second instruction scheduling pass sees the result of new peephole optimizations.

The new construct was created in 1999, but of the 37 back ends present in GCC 3.3, only six use it exclusively. Fifteen still use `define_peephole` exclusively, and six more have both. (Ten have no peepholes at all.) Now, peephole optimization is a relatively minor part of a back end. The majority of the architectures that use either variety define fewer than ten. In terms of code generation, using `define_peephole2` is most beneficial for architectures that use instruction scheduling. The maintainers of any given architecture have no real incentive to update it to the newer style. From a maintenance programmer's point of view, this situation is very bad. The presence of two functionally-equivalent mechanisms for the same basic operation adds complexity and increases the likelihood that something will be broken accidentally.

Peephole optimizations of either variety rarely cause trouble, because the machine-independent code that applies them is small and robust, so it is unlikely to be broken by an unrelated change. However, consider the `cc0` mechanism, which is the older of two possible ways to represent condition codes in a machine description. There are 805 lines of code in the core compiler that are used only by `cc0` architectures, and 79 lines of code used only by non-`cc0` architectures, scattered through 28 files in 121 individual `#if` blocks. This is not a lot of code compared to the total size of the compiler, but it is all in critical places, affecting most of the major optimization passes. Testing on a non-`cc0` architecture will not reveal brokenness in the code used exclu-

sively by `cc0` architectures, or vice versa. The only widely-used architecture that still uses this mechanism[1] is the m68k, and m68k environments are all slow enough that no one wants to test them. It is not surprising, then, that all `cc0` architectures were broken for some time last year.

When someone discovers that a target they wanted to test is broken for some other reason, their usual response is not to bother testing that target anymore. This of course means that nothing stops the target from accumulating faults. By the time someone comes along who wants it to work, it may be easier to start from scratch than to fix all the faults. This is especially true for OS-specific configurations, which break more easily than architectures and require relatively little effort to rewrite from scratch, especially if they are similar enough to the generic Unix that GCC takes for its default.

Recent experience [2] suggests that even CPU ports can age to the point where starting over might be easier. The MIPS back end had not been kept up to date for several years; it was overhauled starting in late 2002, with most of the work done by Richard Sandiford and Eric Christopher. This took six months start to finish, with approximately eight thousand lines of code changed, which is comparable to the effort required to write a minimal back end from scratch. Of course, the MIPS back end is not minimal; starting from scratch might have meant abandoning many of the sub-architectures and operating systems that it currently supports.

A primary driver for the overhaul was the desire to avoid use of the macro instructions provided by the MIPS assembler. This can also be seen as a transition, but not of an API; rather, the preferred style for machine descriptions has changed. When the MIPS port was originally written, the macro instructions were a convenient way to simplify the compiler's job. Now they are seen as a hindrance to quality code generation, requiring awkward workarounds in the compiler.

## 2.2 Functional duplication

Functional duplication occurs when two components both implement some capability that could be shared. A long-standing case exists in the RTL simplification code. When Jeff Law created `simplify-rtx.c` in 1999, he included a comment which gives the flavor of the problem:

> Right now GCC has three (yes, three) major bodies of RTL simplification code that need to be unified.
>
> 1. `fold_rtx` in `cse.c`. This code uses various CSE specific information to aid in RTL simplification.
> 2. `combine_simplify_rtx` in `combine.c`. Similar to `fold_rtx`, except that it uses combine specific information to aid in RTL simplification.
> 3. The routines in this file.
>
> ... It's totally silly that when we add a simplification that it needs to be added to 4 places (3 for RTL simplification and 1 for tree simplification).

It is worth pointing out that at 8,790 lines of code, `combine.c` is the second longest file in the core compiler. Much of this bulk is `combine_simplify_rtx` and its subroutines.

Functional duplication is less likely to cause breakage than incomplete transitions. Continuing with this example, all the RTL simplifiers are exercised by the normal testing procedure, so it is unlikely that one of them will remain broken for an extended period. However, the answer to the question "Why did this bad optimization happen, when I can see that the code in file A is correct?" may well be "because that transformation is duplicated in file B, only with bugs." Furthermore, this duplication invites people to update one set of simplifiers and not another, which means that whether or not an RTL construct gets simplified depends on which optimizer pass encounters it. And, of course, it causes the compiler's runtime image to be bigger than necessary, which contributes to compiler-speed problems by wasting space in the instruction cache.

Law's comment hints at a deeper cause of functional duplication, namely, that we have two different intermediate representations (trees and RTL). In the past, almost all of the compiler dealt exclusively with RTL so this was not a cause for concern. We now do some optimizations at the tree level, and lots more are planned. It would be useful to share

[1] If `(cc0)` appears only in `define_expand` forms that generate no RTL, that machine description does not use the `cc0` mechanism.

code between tree optimizers and RTL optimizers as much as possible. This has already been done for the control-flow graph, on the `tree-ssa` branch. If the data structure holding an expression to be simplified could be made opaque to the code computing the simplification, the same could be done for the algebraic simplification library.

Functional duplication also occurs when a module exists that logically should be responsible for some task, but is not presently capable of it. Instead of fixing the existing module so that it is capable, often people choose to build something new from scratch, which is easier in the short term. A good example here is the language-independent tree-to-RTL converter (`stmt.c`, `expr.c`, etc.) It is one of the oldest parts of the compiler. It still reflects design decisions made when C was the only supported language, and the tree representation was used for only one source statement at a time. When front ends started being rewritten for whole-function tree representations, no one wanted to update the converter to match. Instead, each front end that now uses whole-function trees contains duplicated tree-walking logic, so that it can continue to feed the tree-to-RTL converter one statement at a time.

This duplication not only causes the problems described above, but also hinders conversion of other front ends to whole-function processing, because they would have to duplicate this code again. Nor is there agreement on the form of whole-function trees. The maintainers of the C language family developed one such representation; independently, the Java maintainers developed another, incompatible representation. This prevented the tree inliner developed for C from being used for Java. Rather than copy the file over, it has been heavily `#ifdef`ed, which may or may not be an improvement. (The people working on the `tree-ssa` branch have a major goal of developing a proper, language-independent, whole-function tree representation.)

When a transition is finally completed, or duplicate code finally collapsed together, it may still leave vestiges behind. The garbage collector was completed in late 1999, but most of the obstack allocation scheme that it obsoleted stuck around until late 2000. We are still finding traces of it now, in the second quarter of 2003.

Everyone likes deleting code, so why do vestiges stick around? People usually find vestigial code when working on something else. To delete it, they would need to stop whatever they were doing at the time, construct a fresh CVS checkout, delete the vestige, do a full test cycle to make sure nothing broke, then submit the patch and wait for approval. All this time, they would not be working on whatever they originally planned to work on. We will come back to time consumed by procedures later.

Another reason is, it is hard to distinguish code that is left over from code that was never completed, or that was written in anticipation of a use that never materialized. One can usually figure it out from mailing list traffic or CVS logs, but only with practice. However, no matter what its intended function is or was, code that is not being used now should be deleted; even if a future use was planned, it is likely never to happen.[2] If someone does have a use for a body of unused code in the immediate future, they will undoubtedly say so when its removal is proposed.

### 2.3 Inadequate modularity

Unfortunately, much code that has no apparent function will cause something to break if it is taken out. This is the problem of inadequate modularity. GCC is composed of a lot of logical modules, but the boundaries between these modules are ill-defined and poorly documented. Any given behavior has a good chance of being required by some other module. For instance, the C compiler reads the first line of its input much earlier than would be natural. This is because some of the debugging-information generators want to know what the name of the primary source file is, when their initialization hook runs. These two things may sound like they have nothing to do with each other. But if the C compiler is handed already-preprocessed input, the primary source file is not the file on the command line. It is the file named by the `#` marker on the first line of the file on the command line. Therefore, in order to initialize the debug-info generator properly, that first line has to be read. [3]

The interface between language front ends and the core compiler is especially prone to this sort of problem. This stems mostly from the ad-hoc way in which the front-end interface has evolved. It has never been documented, yet there are seven different languages using it in the current source tree, plus a

[2] This is the YAGNI (You Aren't Gonna Need It) principle.

few more maintained separately. As languages were added, their developers generally tweaked the tree specification around as they saw fit, without much coordination. It was originally intended to cover the needs of GNU extended C only, and still reflects that in some aspects. For instance, the Java front end has interesting kludges in it to cope with the allegedly language-independent `builtins.def`, which is full of C-specific notions like `va_list`. Or, consider the way each back end specifies its platform's fundamental data types: the `*_TYPE` and `*_TYPE_SIZE` macros. These macros map directly onto the fundamental data types of C; if this is a poor match to the language being implemented, one is in trouble. To be fair, most modern platforms define their most basic ABI in a similar fashion, so one might be in trouble anyway.

The interface between the core compiler and a target-specific back end is also very fuzzy. The most basic parts are in the machine description, which is pretty well defined and documented, but there are lots of little details handled by defining macros, which are then visible to the entire compiler, including the front ends. A naive count finds close to five thousand different macro names defined by header files in GCC 3.3's `config` directory. Some of these are internal to one architecture, and some of the headers are not used during the compiler build itself, but there is no easy way to tell them apart. Since the macros are visible to every part of the compiler, every part of the compiler can use them, and does. A target must define almost all of the macros used by the core compiler, which leads to massive duplication.

There is ongoing work to convert all of these macros to data members or function pointers in a global object called `targetm`, which forces a more structured approach. The people doing the conversion are taking the opportunity to clean up the interfaces and create sensible defaults. Thus there is hope that this problem will dwindle as time goes by. However, the conversion project could drag on for years, becoming another of the incomplete transitions that were discussed above. GCC 3.3 has about seventy members of the `targetm` structure; a complete job will require about five hundred, but most targets will not need to override the defaults for most of them.

The core compilers is not free of modularity problems, either. The RTL optimizers are structured as a pipeline of passes, and what each pass does to the code is reflected in the insn chain. On its face that is a modular design. However, there are undocumented limitations to what each optimization pass can handle, which impose constraints on earlier passes. For instance, the first local CSE pass is a waste of time at this point, because the GCSE pass is more powerful... except that GCSE is not prepared to deal with certain high-level constructs that local CSE eliminates, such as `addressof`. This is doubly unfortunate, because GCSE could do a better job than CSE of handling the high level RTL, if it only knew how. [4]

### 2.4 Style

We should not neglect aesthetic concerns. Anything that makes code harder to understand, hides bugs from developers. Anything that makes code harder to restructure, hinders developers from resolving the more serious problems discussed above. GCC's primary failing in this domain is by virtue of sheer size. Particularly in the older parts of the compiler, it is common to find a single function so large and convoluted that a human reader cannot remember all its details. Some may have grown by accretion: `expand_expr` for example may have been much smaller when there were fewer kinds of tree to be considered, or when fewer optimizations were attempted at that time. Others are perhaps stylistically inspired by the "Pastel" compiler that predated GCC 1, which was in a language that supported nested functions; very large outer functions would have been more natural in that language. [5] These functions often maintain state in local variables of an outer block; performing the "obvious" refactor of pulling the inner blocks out to their own functions can cause mysterious failures, since the outer variables are no longer visible.

Gigantic controlling expressions in `if` statements are also common. Here the problem is notational. Such expressions often turn out to be performing pattern matching on RTL, in the most straightforward fashion possible in C. If it were possible to write these expressions in the language used for machine descriptions they would be far more readable.

The macros, idioms, and style constraints which permitted us to build GCC with compilers that predate the 1990 C standard should also be seen as an issue of aesthetics. We already enjoy the benefits of most of standard C's features, such as prototyped functions. However, eliminating all these idioms (as

we can now do) will make it easier to read the code, and this is not a trivial thing. Just the removal of the macros that cloak the differences between traditional and standard C with regard to variable-length argument lists should be a great step forward.

## 3 Procedural hurdles

Once again, let's take a moment and look from 10,000 feet up, this time at the process for contributing a patch to GCC. For this purpose we shall postulate a contributor named Alice, who has a copyright assignment on file, but has not yet been granted write-after-approval privileges, and proposes to fix a bug which appears in the GNATS database.

The first step is to get a copy of the development tree (i.e. CVS HEAD). Then the bug must be reproduced and fixed. The potential difficulties with that were covered above.

Next, Alice must carry out a full bootstrap and test cycle. This is not very hard once you know how. Typical first-time gotchas include configuring in the wrong place or with the wrong sort of pathname, and tripping over a Makefile bug; having the wrong version of GNAT installed, so the Ada front end cannot be built;[3] having the wrong version of autoconf installed, so the configure scripts are broken; and finally, having a broken DejaGNU installation, so the test suite reports thousands of spurious failures. Once all these issues are resolved, Alice gets to sit back and wait for at least two hours. Depending on how slow her computer is, it might be more like a full day. There is also the possibility that the test cycle will fail because someone else checked in a patch which broke the compiler.

Assuming that went fine, the patch is now to be submitted for review. Alice may be ignored for weeks on end, depending on how busy the official maintainer of that component is, whether she has submitted patches before, and how important the bug seems to be. Once someone does get around to responding, there is a good chance that the patch will be torn to shreds and sent back for revision, repeatedly. Alice might get frustrated and give up. If she persists, the patch will eventually get approved. Now (since she lacks write privileges) the person who approved it is responsible for committing it and closing the entry in the GNATS database. If Alice keeps submitting good patches, she will be granted write-after-approval privilege. She can then do these last steps herself.

[3] This is not currently a requirement, but Alice is being thorough.

It is not terribly useful to speculate about the ultimate causes of the procedural hurdles that can be seen in this description. Instead, we will categorize them by nature, as slow or tedious tasks; problems coping with tools; and human error.

### 3.1 Slow or tedious tasks

One of the most important procedural hurdles is the sheer amount of time it takes to develop a patch and get it committed to CVS. Alice had to wait for review, but let's defer that issue for later. Even people with global write privileges are expected to carry out a full bootstrap and test cycle on at least one target, including all languages, before installation. This takes two hours on a 2GHz P4 with 512MB of real RAM, running Linux 2.4. A slower CPU, less memory, or a less efficient operating system will all cause it to be dramatically slower. The author is personally aware of an environment in active use which is centered around UltraSPARC 5 machines running Solaris 2.5.1. On this platform a cross-compiler build, C and C++ only, takes six hours; an all-language bootstrap would take even longer.

On a sufficiently efficient operating system, the bottleneck for a bootstrap is CPU time expended by the compiler itself. This parallelizes well; on a multiprocessor system, `make -j`$N$ will reliably divide the time for bootstrap by $N$, up to some limit. Experimentation is usually required to find the best value to use. However, using parallel make can expose missing-dependency bugs in the Makefile. Since the header dependency lists are maintained by hand, it is easy for these bugs to creep in. Some makefiles have not been written with parallel make in mind; for instance, at the time of writing, `make gnatlib_and_tools` does not work at all in parallel mode. Also, DejaGNU has no ability to run tests in parallel, so the entire test suite must be run serially.

Bootstrap time accounts for the majority of time spent waiting for a computer to do something. However, CVS operations should not be neglected in

this regard. On a higher-end ADSL connection (1.5Mbps down/384Kbps up) a `cvs update` on the mainline takes fifteen seconds—if it has nothing to do, and there are no modified files. If it has updates to download, or potentially modified files that have to be checked (by sending the full text of the file to the server for comparison) it can take substantially longer. Branches are also slower; on the 3.3 release branch, an update with nothing to do and no modified files takes a minute and a half. Recursive commit and diff operations take a similar amount of time.

Once a patch is fully tested, the contributor must write an explanation of the changes made, for the `gcc-patches` mailing list, and a ChangeLog entry. Working out long ChangeLog entries can be tedious. To some extent it can be automated; for example, a simple perl script can extract the names of all the files and functions touched by a patch and format them in ChangeLog style, leaving one to write the "what was done to each" comment, but that part can still be tedious for a long change. This text has to be copied from the message into all of the relevant ChangeLog files, and into the CVS commit log; it is easy to make a mistake along the way.

All of this places a lower bound on the time it takes to develop or revise a patch. Even the most trivial changes have to go through this process, because they *could* have broken something. The time it took to design and implement the change itself is neglected here. That time cannot be said to have been wasted, except insofar as it may have been harder than necessary to make a change, which was discussed above. Of course, the lower bound is only met if the patch works the first time. If the patch causes a regression in some part of the testsuite that must be fixed, then the bootstrap must be repeated.

And the lower bound is only met if the contributor can commit his or her own patches without approval. Otherwise, there will be some time spent waiting for the patch to be reviewed. It is not uncommon to get no response at all to a patch, or even to repeated inquiries. This is not because anyone hates the patch or its contributor. Most often patches are ignored because everyone with the authority and the experience to review the patch is just too busy that week. A lot of GCC's code is listed as maintained by one of the people with global write privileges, or else has no listed maintainer at all. Either way, the set of people who can approve a change to that component is limited to those with global privileges, all of whom are busy. A related problem is that people who do not have authority to approve patches often refrain from commenting on them, even though their opinions are still valued.[4]

Another contributing factor is that some patches are too hard to review. This happens when a patch tries to do too much at once, or when the person who wrote it did not explain its motivation well enough. What seems simple and obvious for the person who was just immersed in the relevant area, may not be obvious at all to anyone else. Splitting patches into minimal changes and explaining them well are both learned skills. At present, we expect people to pick them up by osmosis, but not everyone can learn like that.

Sometimes a patch is not quite right, and sometimes a patch addresses an issue that clearly needs addressing but does not do it in the way that the reviewers would like. When this happens, the reviewers will send the patch back for revisions. Sometimes they send it back so many times that the contributor gives up hope that it will ever be accepted. Then the patch, which might not have been perfect, but was an improvement over the status quo, gets abandoned.

It does happen that patches are ignored intentionally, in order to reject them without having to offer feedback. In most cases, this happens because everyone who could review the patch feels that they cannot have a productive discussion with the person who submitted it. That might be the submitter's fault—there is just no working with some people—but it is much more likely to be a failure of the community. Fortunately this is rare.

### 3.2 Coping with tools

The tools which give people the most trouble on a day-to-day basis are DejaGNU and the `autoconf` family. To begin with the most straightforward issue, the GCC testsuite always produces a handful of "unexpected failure" (`FAIL`) results when run. These failures are *not* unexpected in the standard sense of the word. They do not change often. People who build the compiler on a daily basis and/or follow the `gcc-testresults` mailing list will know which unexpected failures are currently normal for a

[4] This is a variant of the "bikeshed effect." [7]

given environment. They are only unexpected in the sense that DejaGNU has not been advised to turn them into "expected failure" (`XFAIL`) results. Regular contributors are used to this. However, someone who does not build the compiler on a daily basis, or follow the test-results list, will not know whether a given unexpected failure is normal or not. If they are running the testsuite to make sure the compiler works, not having made any changes, they may believe there is something wrong with their environment, or a bug that is not already known. If they have made changes, they will not know whether or not their changes caused the unexpected failures. The only way they can be sure, in this latter case, is to do two complete test cycles from the same baseline code, one without the desired patch and one with. This doubles both the testing time and the disk space requirements, since it is necessary to keep both trees around for comparison.

Failures are not marked expected mainly because it is too awkward. At the least, it involves adding special tags to files in the testsuite. For test cases in the `c-torture` framework it involves creating special files containing snippets of Tcl code. What the tags or snippets should be is mostly undocumented. People usually do it by copy-and-paste from another test case. Further, DejaGNU's ability to describe the situations under which a failure is expected is quite limited. For instance, there is no way to specify that a test will fail if the necessary locale definitions are not installed, or that a test may sometimes (depending on system load) take so long to run that it times out.

There is also a general assumption that expected test failures are not going to be fixed anytime soon, whereas unexpected failures have someone looking at them right now. This discourages people from marking tests expected to fail, because they might be fixed soon and then the marking would have to be undone. Yet tests continue to fail "unexpectedly" for months on end.

If one does not have access to a hosted system for an architecture, one can still test some patches that affect it by building a cross compiler to a simulator target. The GDB source tree includes simulators for many popular architectures. It is easy to construct a combined tree including gcc, binutils, the simulator, and a minimal C runtime, in which to test the cross compiler. However, DejaGNU is prone to glitches when used with a simulator target. One common problem is complete failure to find `stdio.h` or `crt1.o`. One suspected cause of this is invoking `configure` by relative instead of absolute pathname.

Autoconf, automake, and libtool have all undergone backward-incompatible revisions in the past few years. One must have exactly the right version of each installed in order to regenerate GCC's `configure` scripts or Makefiles. For instance, all of the configure scripts presently require autoconf 2.13, which is the oldest version in common use. It is old enough that it is left out of the default installation of some newer operating systems, such as Red Hat 8.0. Use of a newer version might cause visible errors when the script is regenerated or run, or more insidiously it might just cause a small handful of features to be misidentified. Since GCC's Makefiles will automatically attempt to regenerate configure scripts that are older than the parent `configure.in`, a user may discover that the first build from a fresh working copy succeeds, but all subsequent builds mysteriously fail. Using the `contrib/gcc_update` script can prevent this problem, but it will not help someone who has modified the configure script.

It is harder to get in trouble just by having the wrong version of automake or libtool installed, because these tools are only run on specific user request. But one may still be stuck with no way to regenerate files under their control. The author has resorted to updating a generated `Makefile.in` by hand on several occasions.

### 3.3 Human error

From time to time someone checks in a patch which renders the tree unbuildable. Normally it worked just fine for the person who tested it, but breaks in a different environment. The problem may be target-specific, or involve only a language which is not supported by the tester's platform. Or perhaps the patch that was tested is different from what checked in, somehow. Whatever the cause, when this happens, everyone who did a `cvs update` just before starting their bootstrap cycle gets to wonder whether it was their changes that broke the tree.

A few years ago, a CVS checkout taken at a random point in time had a 34% chance of being unbuildable. [6] This is directly attributable to the two-year lapse between the 2.95.0 and 3.0.0 releases. During that time, latent bugs were continually introduced,

until any given checkin had a good chance of triggering one. There was no concerted effort to flush these bugs out until the situation became dire enough to hinder day-to-day work. Since the institution of the three-stage development process, in mid-2001, unbuildable CVS checkouts happen only rarely, since the tree is regularly stabilized.

The automated testers operated by Geoff Keating, Phil Edwards, and others have also been instrumental in reducing the incidence of unbuildable source trees. A failure report from one of these testers can be trusted to indicate a genuine problem—no risk of a quirky environment causing issues—and conveniently lists all of the changes that could have been the proximate cause. They also make people aware of bugs immediately, rather than several weeks down the road when they no longer remember the details of their changes. Unfortunately, at present only a few platforms are monitored in this fashion.

Nowadays, build failures are usually addressed immediately, but testsuite regressions tend to linger for weeks on end. The author believes this is largely a matter of perception. Test cases are often contrived rather than reflective of real code, and the failure may seem unimportant. For instance, at the time of writing, half of the unexpected failures in the C testsuite for GCC 3.3 were caused by incorrect warning messages. Nonetheless, a general habit of ignoring persistent unexpected failures is not good practice.

## 4 Conclusion

Contributors to GCC face both technical and procedural challenges. These can be narrowed down to a short list of causes: incomplete transitions, functional duplication, and inadequate modularity; slow or tedious tasks, coping with tools, and human errors. Some of these problems are easy to address immediately, while others will require long-term, concerted effort. This paper limits itself to discussion of the problems. However, we are confident that solutions can be found.

## 5 Acknowledgements

This paper is largely based on my personal experience fixing bugs in older versions of GCC for a CodeSourcery client. I am also indebted to Neil Booth, Eric Christopher, and Richard Henderson for sharing their experiences. Michael Ellsworth, Kristen Hrycyk, David Johnson, Mark Mitchell, Jeffrey Oldham, and Nathan Sidwell were kind enough to comment on drafts.

Inspiration crystallized around the following IRC exchange between Phil Edwards and myself:

<**pme**> Every time I read *Snow Crash*, I wonder what a GCC "room" in the metaverse would look like.

<**zwol**> Take an H.R. Giger painting, you know, with the perverse and insanely complicated biomechanical constructs.
Now, instead of being all shiny and new, make it old and rusty and overgrown with weeds. Slimy weeds.

A *Snow Crash*-esque view of GCC's code wasn't really what Phil meant, but I would still like to thank him for sparking my imagination.

## References

[1] David Wheeler, "SLOCCount, a tool for counting physical Source Lines of Code." `http://www.dwheeler.com/sloccount/`

[2] Eric Christopher, personal communication.

[3] Neil Booth, personal communication.

[4] Richard Henderson, personal communication.

[5] Richard Stallman, "The GNU Project." `http://www.gnu.org/gnu/thegnuproject.html`

[6] Jeffrey Oldham, "March gcc 3.0 and 3.1 Bootstraps Fail 34% of Time." Email message dated 30 March 2001. `http://gcc.gnu.org/ml/gcc/2001-03/msg01319.html`

[7] Poul-Henning Kamp, "A bike shed (any color will do) on greener grass." Email message dated 2 October 1999, as quoted in the FreeBSD FAQ. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/faq/misc.html#BIKESHED-PAINTING`