Hackers are from Mars and Corporations are from Venus

Michael Meissner
IBM
meissner@linux.vnet.ibm.com

David Edelsohn
IBM
edelsohn@us.ibm.com

Abstract

Hackers and corporations do not always talk the same language and often times have conflicting goals. It can be difficult for corporations to work in the free software space to achieve its goals and just as difficult for the hackers to interact with the corporations.

In this paper, we will cover various problems that often arise due to these culture clashes, and suggest strategies for overcoming the problems.

1 Introduction

Over the last 20 years, we have seen the same dance over and over, of corporations or individuals trying to get changes that they want into the Gnu Compiler Collection (GCC), the binary utilities (assembler, linker) or debugger (gdb), and getting frustrated at the process. At different times and places, we have been on both sides of the table, as an employee of a corporation trying to get our changes in, and as a maintainer for the Free Software Foundation trying to improve the toolset and telling people to change their code. It is kind of tiring to see the same things happen over and over again.

The points we offer in this paper are not magic bullets guaranteed to get your code in without any problems, but suggestions as to ways to make the process less painful. If you look up our posting history with patches, you will see that even after all of this time, we often times have complaints about the patches, and have to go back and rewrite patches to make things acceptable.

It is likely that many of the people attending the talk at the GCC summit probably have already been through the hurdle of getting their code into the toolchain. But everybody has to start sometime, and this paper hopefully will talk to the people contemplating adding their changes.

We also hope the paper is helpful to explain to management of companies contemplating contributing what is involved. It is better to know what the road is going to be like ahead of time, rather than rushing in and finding things are different than you expect. For example, over the years, we have had several different managers, and we have had to explain the ins and outs of contributing to the FSF each time.

1.1 Disclaimer

Generally disclaimers are implicit, but this talk is not a standard technical talk, and it is important to actually state the disclaimer in print. The views expressed in this paper and talk are that of the authors, and do not necessarily reflect those of our current or previous employers. We are hackers, and not lawyers, so seek our your own lawyer for interpretation of the finer points of the GNU pulic license (GPL).

1.2 What's in a name?

In this talk, we call the people who work on free software projects "hackers", which is in the context of the free software community means developers. It does not mean people who try to break into systems, plant viruses, etc.

We will also use the term toolchain to refer collectively to all of the elements of compilation owned by the Free Software Foundation (gcc, assembler, linker, disassembler, debugger).

1.3 Stereotypes

One of the reasons we are writing this paper is both sides tend to have stereotypes of the other. It can be easy to fall into the trap where corporations view hackers as anti-social technology nerds, and hackers view corporations as soulless entities trying to stamp out individuality. The problem with stereotypes is they can get in the way of actually getting what you want done if you feel the other side is morally objectionable. At the end of the day, we are all people, and we view the world from many different perspectives.

1.4 People vs. organizations

- One of the stumbling blocks that the two cultures run into is people vs. organizations. In general, due to their nature, corporations, tend to view things in terms of organizations, while within the free software world things are organized by individuals. This can be a stumbling block, because there is no management to go to, and instead the corporation has to deal with individuals, all of whom come to the process with different motivations and skill sets.

- For a corporation, this means the best strategy is having people who interact with the free software community on an on-going basis, since the free software community is people based. If you rotate people into a slot to deal with the community and then rotate them out again, there isn't much continuity, and often times the new people have a hard time being heard. Most developers are more likely to respond to people that they've had previous dealings with. Now, this can either be working with developers who are on the inside track, or it can mean hiring your own people to interact with the developer community.

- Similarly, most developer communities do not defer to "experts". Someone may have designed and implemented an important feature in a company product or have an important title in the company or be an academic expert in a particular topic, but that does not mean they understand the community's codebase or priorities. Deploying a company "expert" to tell the community what it should do is unlikely to be effective. If the person participates in the community and demonstrates that his implementation is best, the community will incorporate the contribution.

- Another difference in approach is hierarchy. Most free software development is very distributed with leaders and gatekeepers who have developed expertise in various areas. Large corporations, on the other hand, have centralized authorities "architects" and "design councils" who create the design and plan and give out marching orders. The leader or leaders of a free software project rarely can direct the community to develop or accept a feature proposed by a corporation. One must convince the gatekeeper responsible for that domain.

- A lot of the issues in getting patches approved are a variant of he said/she said type issues, where there is no true communication going on. It may help to step back, and more clearly explain what the problem is that you are trying to solve, rather than assume the other party knows everything you do. GCC is a very complex beast, and it runs on a lot of environments, and most people do not know the ins and outs of every single platform.

- In the past there was a phrase called "Egoless programming". In general there is no such thing, but it is important to realize in dealing with another person, that either your ego or theirs might be getting in the way. They may be right and you might be wrong (of course it can also be you are right and they are wrong, or some mixture in between). A lot of the fights in computer developer circles is due to ego (and this applies both within the free software community as well as within development groups inside of corporations, but you don't always see the battles for non free software groups).

- Even if you think the other side is wrong, it generally works better if you keep things at the purely factual level, and keep your own ego in check.

- Sometimes however, you can get in such a rut that no forward progress can be made. It may make sense to replace the employee with a different person who has no history, can get things started again. Often times being different means that person doesn't have as much stake involved in the code as it is, and might be willing to make changes.

- In photo sharing forums, you often read complaints from people who are new and put up a picture for comment that *why does Jane Smith always get comments when she posts, but nobody looks at my picture*, which is similar to the *I posted a patch and* *nobody looked at it* comments that are heard every so often. If you look at the posting history, the person who gets lots of comments, tends to be the person that spends more time interacting with the other people, and people are commenting because they know the poster. The same thing tends to happen in terms of patches.

- Another analogy to the photo forums is if you post a picture with a subject line that doesn't describe the photo many viewers will skip over the picture. In the same fashion, if your subject line for a patch is something generic like just the bugzilla number, or says crash, many reviewers who are short on time will skip over it.

1.5 What hat is a developer wearing

Developers in the free software community often times wear many different hats. Sometimes they are speaking for themselves, sometimes they are speaking in the context of the maintainer of a certain part of the compiler, and sometimes if they are employed to work on free software, they might be speaking for their employer. People new to the process should try to figure out where the other person is coming from and what hat they are wearing.

Similarly if you are wearing different hats, you should indicate which hat you are wearing if it matters. This can be important, especially if you post email from a corporate email address, and most posters would assume you are speaking in behalf of the corporation.

1.6 Motivation

One thing that seems hard for companies to come to grasp with is motivation. In general, corporations tend to deal with motivations that are purely monetary based (*I will pay you this amount of money for these services or goods in exchange*). In the free software world, people will work on free software for other motivations in addition to monetary, such as wanting to optimize their particular application, working on a cool project, or to run on their particular piece of gear.

A lot of times, corporations do deal with this in terms of managing employees where the good manager tries to match the jobs to be done to the people being managed, but they don't do this with external entities. Spending some time to understand where the other people are coming from and what problems they are trying to solve, might be able to minimize the differences of opinion.

Another thing that can be helpful is to realize that not all free software developers are the same. We each have different motivations and motives, and one size fits all approaches often times backfire.

1.7 Free Software Foundation

The Free Software Foundation, it was established on what it calls the four freedoms `http://www.gnu.org/philosophy/free-sw.html`:

- The freedom to run the program, for any purpose;
- The freedom to study how the program works, and adapt it to your needs;
- The freedom to redistribute copies so you can help your neighbor; and
- The freedom to improve the program, and release your improvements to the public.

The upshot is if you intend to modify the code owned by the Free Software Foundation and share it with other people, such as the compiler or binary utilities, you need to follow the rules that they have established. You are free to write your own compiler and binary utilities, just don't use the code that is part of the toolchain in ways that violate the license agreements.

You are allowed to do your own modifications to the toolchain without restriction as long as you don't distribute it in either source or binary form. The restrictions that the FSF puts on its software are meant so that if you modify the toolchain and distribute it, that you need to distribute it in a fashion that allows other people to be able to modify your changes and places no restrictions on their downstream use of the toolchain.

Sometimes you run into people with the idea that since GCC is free software that they can do whatever they want with it. It is not public domain software, it is under copyright. The owner of the copyright (FSF) has decided to place some restrictions on the software so that it furthers the goals that it was founded on.

1.8 Why go through this trouble?

As painful as getting through all of the hoops to contribute code is, there are good reasons for companies to contribute to the effort:

- **Customer requirement**: Some users require using GCC over other compilers, and if there is no support for a particular hardware setup, these users will go elsewhere. The cost of doing hardware enablement needs to be balanced against the possibility of lost sales.

- **Time to market**: A hardware company has prototype hardware available years or months before it is available for sale to the general public. If you wait for somebody to add the feature on their own, it may take years or decades for other people to write the support. The hardware may even be obsolete before somebody else adds the support.

- **Optimization and tuning for hardware vendors**: In general new computers that are in the same family as previous computers have somewhat different tuning parameters, and the hardware company has the best insight as to the best code generation strategy to use.

- **Optimization for applications**: Even if a company does not sell hardware, it may have a core application that it wishes to run as fast as possible, and paying for people to add optimizations that are not being done can give it a competitive advantage.

- **Board evaluation**: Embedded companies often times tailor GCC to support their evaluation boards with GCC, and put it out for free to convince people to buy their chips.

- **Linux and BSD**: GCC is the compiler of choice for Linux and BSD operating systems, and if a hardware vendor wants to sell into that market, they need to provide support for their hardware if there are appreciable differences in the hardware.

- **Giving users options**: No user likes being locked into an environment with no chance of switching. Even for companies that offer their own compilers, supporting GCC as a second compiler, gives the user a choice.

- **Bug fixing**: Even if a user never needs it, having the source available to GCC means that in a critical event, a user could patch the compiler on his own to work around a critical bug.

More information can be found at: `http://gcc.gnu.org/contributewhy.html`.

2 Legal stuff

One of the issues that corporations and individuals run into is the various legal issues in contributing to the Free Software Foundation. As we said above, we are not lawyers, and anybody should consult with their own legal experts on the precise meaning of the various licenses. These thoughts are things we've observed over the years, but it is not intended to be an in depth exploration of the Free Software Foundation licenses.

2.1 GPL, its not just a good idea, it's the law

Every so often, companies run afoul of the **GPL** (Gnu public license) or **LGPL** (Lessor Gnu public license) that the toolchain and libraries are distributed under. Sometimes companies are intentionally violating the license, but a lot of times things happen because of invalid assumptions. The Free Software Foundation does protect its copyright of the toolchain sources, including using the law to enforce the requirements.

- If you release a set of toolchain binaries, you need to release the source that goes with it. Usually the best strategy for a corporation is to release both the source and the binary at the same time, and on the same media.

- If you release a set of toolchain binaries but not the source at the same time, you need to be able to release the sources for at least three years after you released the binaries, and longer if you are offering spare parts or customer service for hardware covered by the binary release. In practice this can be hard to do as a company has to guarantee that it will still provide the source, even if it was bought by another company. It is much easier to meet the requirements if you release the source at the same time as the binaries.

- You can distribute changes to the toolchain independently without trying to get the changes contributed back to the Free Software Foundation, but the changes must follow the varioius licenses. Typically, it is better to get the changes merged to the official repository, rather than maintaining your own patchset, but it is an option.

2.2 Contributing to the GNU compiler collection

The main rules for contributing to GCC are noted at:

- `http://gcc.gnu.org/contribute.html`
- `http://www.gnu.org/prep/maintain/maintain.html`

.

In terms of companies contributing to GCC, some things to keep in mind are:

- In a work for hire, a company can decide that the developer assigns rights to the company, and then the company donates the code to the Free Software Foundation, or the company can disclaim the rights to the software, and have the individual developer contribute the changes. We have worked in both situations, and generally the first option is the best.
- Unless you know you are going to be doing only one donation, it is better to setup the assignment for all future changes.
- In either case, somebody who has signing authority for the corporation must authorize the assignment. A lot of times, this can be a crucial step, in that the people doing the work need to present a business case why this assignment is needed.
- Sometimes the legal staff needs to be educated as to why doing this is a good idea, and that you aren't going off on a wild goose chase.
- Until the paperwork is signed, you can't get your changes into the Free Software Foundation repository. It happens frequently that work has started on a port without the developers having the ability to get the patches committed.

2.3 GPL v3 vs. GPL v2

In November 2007, the Free Software Foundation updated its main license (**GPL**) to version 3. This paper is not the place to go into the various differences between version 2 and version 3 of the GPL. Here is the site that discusses the license: `http://gplv3.fsf.org/`.

Some corporations have decided that they cannot adhere to **GPLv3**, and will stay at **GPLv2**. In the context of the toolchain, all active code development is now done at **GPLv3**. If a company's legal staff feels that they cannot comply with the **GPLv3**, then they cannot contribute to the active development of the toolchain. This also means it limits the pool of developers who are willing to work only on older toolchains and not be able to contribute their work back to the Free Software Foundation.

2.4 Disclosure of future machines

One of the things that comes from time to time is when a corporation discloses new hardware changes. Because the toolchain source is freely available, developers cannot put support for new hardware into the Free Software repository that are enabled with a new switch until some of the details of the new hardware are announced. Typically this decision must be made by senior management of the company because this is seen as giving the competition advance warning.

This usually means that developers must work with their management, so that the announcement is made early enough that the changes can be put into the development sources. Often times, this pre-announcement is in the form of an update ISA (instruction set architecture) that describes new instructions that the compiler would generate, but not a lot of of other details for the chip (cache size, processor speed, cycle speed, number of cores and processors, etc.).

3 Doing the work

Once you have cleared the legal hurdles, then you can start contemplating making changes to the toolchain. It out of the scope of this paper to discuss all of the issues in modifying the compiler, etc. Instead we will leave it as an exercise to the reader, and assume that you have code that you want to contribute back to the Free Software Foundation.

3.1 Release schedules

If you are making a change to work on certain hardware, you need to keep abreast of the state of the release cycle (http://gcc.gnu.org/develop.html). In general, there are several stages that the code goes through for GCC (the binary utilities and debugger have different cycles):

- Stage 1 – large features can be added
- Stage 2 – smaller features can be added
- Stage 3 – bugfixes only

GCC is starting to evolve towards a Stage 1/2, Stage 3, Stage 4 development model, diminishing the distinction between large and small features, while focusing more on bug fixes before (Stage 3) and after (Stage 4) the next release is branched but before it is released.

If you are adding support for new hardware, you should be aiming to contribute it in Stage 1, and if you miss the stage, you will need to wait until the next release.

One thing that comes up from time to time is the requirement that supporting new hardware is considered a new feature and not a bug fix. This means it may be a year or so before you can add support to the official tree. Common strategies for dealing with this are providing an advanced toolchain either directly or through a third body, like a university, or getting the Linux/bsd distributions to support your machine.

3.2 Keeping your code up to date

- It is often tempting to stay with a given version of the compiler while doing development so that you minimize the impact that external changes made by other developers. However, before you submit the changes, you will need to merge up to the current version of the sources. In the long run, it is less painful if you do the merges every so often, and fix the problems incrementally, rather than trying to do a big bang approach.
- One way to do this is to maintain two branches of checked out code. On branch is the mainline with no changes, and the other is the development branch. There are various scripts out there that help you do these merges automatically, such as a script called svnmerge (`http://www.orcaware.com/svn/wiki/Svnmerge.py`) that uses subversion to manage the merge step. When you are satisfied with the changes on the development branch, check them all into the branch. Update the mainline branch, and do a svnmerge step on development branch, and bootstrap/test both branches. If there are merge conflicts or errors in the test, commit the changes to the development branch and repeat the cycle.
- Most times, everything works automatically, but there are times when you will need to modify the code to take into account the changes to the mainline. It is a lot easier, when the amount of changes are smaller, than say two or three months worth of changes.
- If you are working on machine ports that have not been disclosed yet, you could maintian a subversion repository on a machine within your company. The main branch of the repository would be pure mirror of the mainline branch, and you would maintain a branch off of that to do development. When you want to do a merge, you would check out the branch from the Free Software development machine and then check the sources into your local repository. This way you can update and merge with the mainline at any time, while still maintaining the work you are working on.
- It is good practice to keep a private ChangeLog of the changes on the development branch, which keeps a running commentary of what you have done. You probably want to use a file named something other than *ChangeLog* for this private branch, so that the normal *ChangeLog* file is not touched.

3.3 Backports

Generally, compiler developers are one or more releases ahead of the users. Sometimes users will not consider moving to new versions of the compiler because they do not want to deal with the changes that the rest of the compiler will introduce (new optimizations, changes to language semantics, etc.). This means that if you are providing compiler support for a new piece of hardware, you probably need to plan to add this support to older versions of the compiler. In general, it will not be just a drop in change, but you may need to rewrite parts by hand, and hand merge these changes in. The further back you go, the more painful this becomes.

For instance, as we write this paper, we have a backport of power7 changes added to branch based on a 4.3 GCC, and it has some subtle failures that need to be identified and fixes.

3.4 Private features

Some companies have created features in their version of the toolchain that is never merged back to the mainline. Sometimes these features are rejected from going in, sometimes they are done in a different fashion, and sometimes the features are never proposed for inclusion into the mainline. In any case, having private features can add significantly to the cost of maintaining the port, and it means your users are locked into using the alternative toolchain. It is the best practice for all code to be pushed to the mainline to reduce the cost of maintaining parallel toolchains. Sometimes this is needed to run an existing code base, but you should strive to make it rare.

Using private features often times locks you to a particular revision of the compiler that those features were developed on, particularly if the features were developed by a third party, and not with in-house staff who can move the code forward. Over time, it becomes harder to find people who are willing to keep this old code up and running, and supporting this code becomes more of a financial drain.

3.5 OS distributions

- Most users of GCC do not build the compiler themselves, nor do they necessarily download the compiler from a support site. Instead they usually rely on their OS distribution (Linux, BSD, cygwin, etc.) to provide a compiler. Some distributions see themselves as providing cutting edge tools, while others see themselves as selling stability, and update their compiler toolchain to new revisions only when the distribution goes to a new major version number.

- This means that it might take the average user years before they get to the compiler being developed right now (4.5 is being developed at the time this paper was written), factoring in users not upgrading their system and OS distributors only changing compiler base revisions once every 1-2 years.

- Companies that offer new hardware more frequently than that, need to have plans to get the distributions provide support for the new hardware in either their main compiler version or an advanced compiler version. Usually this is one or two versions back, which means backporting the changes from the development branches to vendor branches. Each OS distributor uses its own branch, which multiplies the number of branches that need to be tested.

- OS distributions usually require that the changes be merged upstream to the mainline development branch before accepting a backport. This can be problematical when the development tree is closed to new features.

- Usually what happens is a series of negotiations with each of the OS distributors of what patches to put in their next release. This multiplies the problems facing corporations with new hardware, since it means several different groups to make the case for the changes, and you need to spend a lot of time maintaining various trees.

- One thing that is a problem is often times hardware companies limit their involvement to just the top distributions. For example, in the Linux world for business computers, this would be **Red Hat** and **SUSE**. Now for things like mainframes this likely makes sense, since you don't just go down to *Best Buy* and buy a series z mainframe, but for the x86 desktop/laptop world, it may be useful to also talk to the other distributions like **Mandrivia**, **Ubuntu**, etc. It becomes a job to balance the cost of dealing with multiple distributions vs. the payback in terms of not alienating potential customers.

3.6 Making it general

- One of the things that developers working for hardware companies tend to not think about is making the code that they are working on general enough for other platforms to use. If you are only modifying the machine specific files and not adding new machine dependent passes, this may not apply. If you are adding new optimizations in the machine independent parts of the compiler, you should strive to make your solution general enough for other ports to use.

- Usually this means soliciting feedback in the mailing lists or on IRC for other developers, and see if they have similar needs, and then working with them to come to a common goal. For example, in the named address space support that is the subject of another paper, the machine in question only needed one extended address keyword. It didn't need the full generality of having multiple named addresses, but we added the more general support called out in the report that described named address because it might be needed for future ports.
- One ironclad requirement is in the machine independent section of the compiler, you cannot call into machine specific functions directly, but you must use the target hooks facility to do callbacks. If you only test on one platform, you would not notice that you are calling machine specific functions.

3.7 Fixing problems and not symptoms

GCC supports a wide number of architectures, ABIs, OSes, and languages. This improves code coverage and testing, but also makes testing and bug fixing more complicated. Where one could paper over a problem in a compiler with a more limited target, the design of GCC necessitates and the community enforces fixing problems, not symptoms. Sometimes the compiler has a segmentation violation or other internal error. Usually this is a problem caused earlier in the code, and may only happen for some machines. It can be tempting to just put an if statement to avoid the particular situation that is failing, but it is often times better to do research on why the thing was failing, and fix it upstream if possible.

3.8 What chipset to tune for

The maintainers for toolchain ports that span multiple implementations of the basic chipset, like the x86, power, mips, and arm platforms have to decide what are the characteristics of the tuning parameters for the default machine if the user did not specify to compile for a specific machine. If a chipset is made by several different companies, then there is a delicate balancing act as to which defaults to use. Even if a single company makes all of the chips, the maintainers have to decide whether to favor the machines that are currently out in the field, or the next generation of machines. In general at every major release, the maintainers should go back and look at the defaults, and in general favor the newer machines over the older ones.

Another area that comes in is maintaining tuned libraries for each chipset. In general, there is a tension between having libraries for every single option, and the whether this leads to exponential in the size of the release and lengthening the time it takes to qualify a product on each different variant of the hardware platforms. Hardware companies tend to be on the side of tuned libraries while distributions tend to be on the side of a single library.

3.9 Testing

It is good practice to do tests as much as possible so that the number of changes is smaller, and easy to isolate the bugs. As we mentioned earlier, you should merge to the mainline and test on a regular basis. However, merging up to the mainline has its costs, particularly when the compiler is in stage1 development, as mainline can be broken for some periods of time.

One practice is to fire off a build/check of the mainline source that the development branch is merged up to, and compare it to the development branch to see whether there are regressions. If the mainline fails, then you need to investigate whether it is a general failure, or whether it is just your particular environment that fails. If the branch fails, then you need to figure out where you need to improve your code.

In addition to testing the code on your own platform, if you have touched machine independent files, you need to test at least one other platform to make sure that your changes do not break other compiler ports.

3.10 Simulators

- When working on a new chip variant or new machine, work is usually done on simulators before the real hardware is available. The problem with simulators is that they are slow. Ideally you will only run the binaries for controlled tests on the simulator. If you run the usual *make check* under a simulated machine and run the compiler in that simulator, it may take months to get one run done for the simulator to simulate the compiler compiling thousands of test cases. One solution is to make the simulator run an OS with a simulated network device, and use the remote support that exists in the dejagnu test infrastructure to run the compiler locally, but send jobs to the machine. Another approach is to invoke the simulator each time to run each command.

- For a lot of work, a simple instruction set simulator will allow the compiler writer to test code generation much quicker than using a more complex simulator that models the whole system enough so you can boot an operating system and/or model pipelines and cache states. For tuning you might need to use the more complex simulator.

- One thing that is important is that the simulator and compiler both be validated against the actual hardware. For instance, one of us did a port to an embedded machine via a simulator years ago, and both the person writing the simulator and compiler mis-read the manual in the same way (whether 16x16->32 multiply sign extended or zero extended). The upshot was the compiler was compatible with the simulator, but it was not compatible with the hardware.

- If you are using Linux as a development system, you can register to use a simulator based on the magic numbers in the binary via **/proc/sys/binfmt**, so that for the purposes of running the testsuite, it looks like you are running a native executable.

- If you don't have a good simulator available and the hardware is not yet available, consider writing a simulator based on the ones in the *gdb* package. The advantage of using those simulators is they provide a debugging environment in addition to pure instruction simulation.

- Another advantage of doing the gdb simulators is it lets other people run your code on their systems. This can be a helpful for instance if you have a question and need somebody to help you debug your compiler, or if you want to split doing development among different people or organizations.

3.11 Bringing up GCC on new hardware

- In terms of corporations and new hardware, one issue that comes up is how much access compiler developers have to test their code on the real platforms. Usually, these machines are fairly scarce and there are a number of people in the company trying to get access to the machine. If the development is being done by an outside group, it can be even more problematical to get enough access to the machine, particularly if the chip is late. This is a management issue, and is one of the things that corporations need to include in their plans is enough time for testing.

- Another thing that comes up is that alpha/beta hardware may undergo several cycles as the hardware is debugged. It is important that compiler developers be given access to the newer chip versions as they become available. Sometimes the changes are changes in what instructions are available (and the encoding), the instruction speed or cache policy between the alpha version of the chip and the chip that eventually reaches the customer. If a toolchain is optimized for the alpha chip, it may need to be re-optimized for the final chip.

- In addition to timing differences, alpha hardware often times has bugs in it, and a compiler team is called on to make the compiler not do some instruction sequences for alpha chips. One of GCC's strengths is that it can easy to add such tweaks in a timely fashion and it would allow for other software to be tested on the machine until the next version of the chip comes out that fixes the bug.

3.12 Using GCC to perform what if type experiments

One area that hardware companies don't use as much as they could is a collaboration between the compiler writer, machine architects, and simulator writers to explore what if type possibilities. For example, you might want to consider whether it is profitable to add a floating point multiply and add (*FMA*) instruction to the architecture. A private version of GCC would be made to add the *FMA* operation, the simulator would be modified to add *FMA*s, and then all of the usual benchmarks would be run to see how frequently the *FMA* is used, compared to the cost of adding it to the next generation of machine. If it doesn't pan out, you go on to the next test, and you haven't spent much time in terms of finding this out compared to the cost of designing and fabbing a chip. Modifying GCC in this way is usually faster than a native compiler, since usually GCC already has such support for other ports, and it is a matter of getting the right definitions in the machine description.

For example, the *FMA* case is pretty easy to do in GCC, and it took a very short time to do. During the investigation, the actual instruction changed several times, and it was a simple matter to change a private version of GCC to track these changes. GCC was used to experiment with the PowerPC architecture before the ISA was finalized and GAS was used to experiment with the AIX 64 bit XCOFF file format.

By having the compiler staff early enough in the design stage, it can avoid adding instructions that are hard to optimize for in the general case.

3.13 Embedded ports

If you are developing the compiler toolchain for an embedded environment, it is useful to have a complete setup that includes a stub library, so that you can run the testsuites before you get a real chip made. Having a public simulator means that it is possible for other people to build and check for failures.

For embedded compilers, one development method that is done (sometimes with multiple people in parallel) is:

- Do the GNU binary utilities port for the assembler, linker, and disassembler.
- Do enough of the compiler port to start on the libraries.
- Add support for a simulator of the machine with system calls to do read, write, and exit.
- Port newlib or similar small library to work in the simulator to provide the full library that uses your stub system calls.
- Start work on the debugger.
- Finish up the compiler port.
- Contribute each of the changes back to their respective repositories.

3.14 Mailing lists, IRC, and firewalls

In the old days of GCC development, all work was done via the mailing lists (`http://gcc.gnu.org/lists.html`). Now in addition to the mailing lists, there is the **#gcc** channel on the internet relay chat **irc.oftc.net/6667**. Many of the developers hang out there, and it is useful to ask quick questions. Often times, questions are answered within minutes of being asked, that never go to email these days. If you are not able to use IRC, you are cutting yourself off from a valuable resource, and making your job harder.

If you use corporate mail systems like Notes or Outlook, it can mangle patches, and make it hard for people to communicate with you. Some people use mail systems like gmail for their mail, rather than their corporate mail. However, some companies prevent employees from using such services, and it can be a nightmare for people to do development with such mail systems. We have seen people have to leave the office so that they could get access to their mail or ftp from their home system to sites that might be blocked by the corporate firewall.

One thing to watch out for is whether your mail system is robust enough to receive the amount of mail in the mailing lists, and that you have enough space in your mail quota so that you don't have an interruption in mail service. If the free software foundation mail servers get too many bounces, you will be removed from the lists. It can be really hard to track down why mail is disappearing when you work for a company with a large infrastructure. Another thing to check is whether mail from other developers is being considered spam and not being delivered to your inbox.

Developers need access to the machines that the sources are on to do checkouts and checkins of the source. This may involve setting up a hole in the firewall for accessing the gcc.gnu.org machine over port 22 (ssh) or it may involve having two computers, one on the internal network, and one on an external network that does not include anything that is confidential.

IRC and chat is another thing that is often times blocked by corporate security. At one company, IRC was blocked by company policy, and the GCC group had to get special access to get to the GCC IRC channel. This took a lot of justification to get this access. The point is the Gnu compiler is a joint effort among people throughout the world, and if a company hampers its employees from participating in the development, they will be marginalized and have a harder time to get their code accepted.

3.15 Big bang vs. incremental changes

If possible, it is better to do changes incrementally rather than as a whole big change set. The reason is it is easier for the patch reviewers to grasp what you are doing as a whole, rather than trying to understand a couple of megs of changes. However, if the development tree is in stage3 for a long time, these patches build up.

Now to be honest, a lot of work is done in big bang style, because of the patches building up. Also, the code can vary quite a bit as the developers are trying out different things. In fact as we write this paper, we have two major branches (named address spaces and power7) that need to be merged. Having a whole ton of patches to contribute all at once is going to make getting this code in harder. Usually, we try to make smaller patch sets to reduce the load on the reviewers, but sometimes when you make a lot of changes, it is hard to make small patch sets.

3.16 Getting help

It can be hard coming up to speed on working on the GCC compiler. There are various how-tos and documentation to read, but once you've read the documentation, it can be useful to ask people questions via the mailing lists or IRC channels. Generally, it is better to ask a specific question, rather than asking generalities. Remember, that most of the experts are very busy, and may not know all of the details of your particular implementation or port. Show that you've done some investigation of the problem rather than just saying *something is broken, please help*.

4 Getting your work committed

Once you've gotten through the legal hurdles and the challenges to actually add a new feature or fix a bug, the next step is to get the code committed to the tree. Sometimes this is a quick process, and sometimes this is an involved process. For example, it took 3-4 years to add support for having an option that the port could tweak to say that int bitfields are unsigned instead of signed.

4.1 Who needs to approve your code

Who needs to approve your changes depends on what areas of the compiler you have modified. Currently GCC has maintainers for various sub-parts of the compiler, and if you modify code in that section, you need the approval of the appropriate maintainer(s), or the global reviewers. Unless you are a maintainer for a section of code, you need approval for changes (which includes the global reviewers for their own changes). If you modify multiple areas of the compilers, you need the approval for each of the maintainers that your code modifies. As of this writing, there are maintainers for:

- CPU port maintainers (x86, arm, etc.)
- OS port maintainers (linux, hp/ux, rtems, etc.)
- Language front end maintainers
- Library maintainers
- Infrastructure and optimization maintainers (gimple, tree, rtl, non-algorithmic, etc.)
- Global reviewers approval are needed in a few cases that are global in scope, like adding a new port

If you are working on a GCC port for a new machine, you should go through the paperwork of being a maintainer for the machine.

4.2 Review cycle

Once you have finished the changes, you need to go through the steps of getting your code reviewed. The details of doing the review is covered in http://gcc.gnu.org/contribute.html. What we want to talk about is ways to make the process less painful.

- Make sure you have your paperwork up to date before submitting patches.
- Make sure the patch applies to the current tree before submitting the patch.
- Make sure you indicate what level of testing you have done, and if the change involves machine independent changes whether you have verified that the changes do not affect other ports.

- As we mentioned earlier try to submit patches in smaller self-contained chunks. All of the reviewers are busy people, and may pass on reviewing large patches, because they don't have the time to look at a large patch in great detail. Note that there are posting limits for patches, and even mega-patches might have to be broken up so that you can submit them.

- Try to understand what the maintainer is saying and see where they are coming from. If you are unsure, ask the maintainers for suggestions, but don't expect them to write or debug your code for you.

- Don't take things personally. Even if you feel a reviewer is being unfair to you, keep things at a professional level when you submit patches. One of the things we have mentioned to people that we are mentoring is that it is helpful if you grow a thick skin and let the little things slide away. If you do feel that things are in a rut, consider trying to get somebody else with less history to work with the maintainer to find an acceptable solution.

- Ping the maintainers on the mailing list or IRC if the patches have sat around for awhile. It would be nice if all patches were acted on immediately, but with the great number of patches going out there, it is very easy for things to fall through the cracks. However, realize that the GCC development is done worldwide, and all of us have lives, so you need to wait a few days before sending out a ping message.

- Don't use Notes or Outlook to submit patches. They tend to do weird things to the average patch, and make them impossible for a maintainer to try out the patch on their own system.

4.3 Coding style

One of the things that often times trip people up is not using the appropriate coding style. Just sit down, and learn the coding conventions (`http://gcc.gnu.org/codingconventions.html`) until it becomes second nature. This includes making sure all lines are less than 80 characters, no space in front of unary operators, and making macros uppercase.

Make sure you include a properly formatted ChangeLog file(s) for each subdirectory that has ChangeLogs. These ChangeLogs should describe all of the changes made.

If you are fixing a bug that is not in the testsuite, add one or more tests to verify that the bug is fixed.

4.4 Who are you? Will we see you again?

- One of the common complaints about the current system is that people post patches and never get a review of the patch. Part of the problem is it can be hard to get noticed. In general, you increase the chances of people replying to you by being doing more work. However, if you are perceived as not knowing what you are talking about, people can sometimes ignore your posts.

- One suggestion is to spend some time fixing bugs is a good way to get your feet wet. Not only does it give you skills in working on the compiler, it makes you more visible, and maintainers often times remember frequent posters.

- One thing that people working for corporations don't always understand is that fixing bugs other than just in your port, can raise your status higher, rather than fixing only bugs in your port. This is a feedback cycle, and as you gain more status, it may be easier to get people to review patches. Unfortunately, it is human nature to only concentrate on you little part of the compiler, and a lot of us don't stick our necks out.

- If you are fixing a bug to get experience, and the bug becomes too complex, go to the next bug and look at that. We have seen people spend a lot of time on bugs like this, and it can be hard to justify spending that much time if it is too complex. One suggestion is to look at bugs that have been sitting around for awhile are likely not being worked on actively by somebody else, as it can be frustrating to work on a bug for a week, only to have somebody who knows that part of the compiler fix it in a small amount of time.

- Going to the GCC summit or other places to meet developers in person can often times help new developers get recognition, which in turn helps getting their patches noticed and reviewed. For senior developers, it can allow them to plan with other developers what to do in the future. In these days of severely restricted travel budgets, it can a hard sell to upper manager to balance the terms of the cost of the trip vs. the expected gains.

5 Working for a corporation

Most of the previous parts of this paper centered on the culture clash between free software developers and corporations from the point of view of the corporation and their employees. In this section, we will cover some of the issues that have come up over the years for free software developers when they work for the corporations. We should note that every company is different, and that these notes pertain to various companies over the past 20 years.

5.1 Getting a corporation to understand and support the GCC toolchain

If a company does not have a strong presence already in the free software world, it can be a really challenging task to get the company to look at GCC. The employees that are proposing to add GCC support really need to develop a strong business case to present to upper management which outlines risks, costs and return on investment. In other words, you need to think like a corporation, and submit these things in the language that the corporation understands. In most companies this involves getting a champion at the executive level who understands the issues and can write the business case. Often times in large companies you need to work up the layers, first convincing your local management, and then middle management, and then executives.

Many years ago, a company was coming out with a new machine using a new chipset. At the time, the company had some serious issues with the compiler vendor they were dealing with, to the point that they were looking around for another compiler to use. A developer came across GCC at this point, and saw that somebody had started a port for the chipset they were using. The engineer mentioned this to his manager, and after some discussion about what was GCC and what really was covered by the GPL, the manager became a champion of using GCC, and was able to convince the senior vice presidents to go with providing support for GCC. Initially it was just thought of as the compiler to ship with their version of UNIX, and that customers would buy the normal compiler from the compiler vendor as an add-on cost.

When the port was finished, GCC became the standard compiler. At one particular point, the development team was told that GCC was much slower at a particular benchmark (**dhrystone**) than the other compiler, and that management was thinking of going back to that vendor. After some checking, it was discovered that the other compiler optimized the string copy (*strcpy*) of a constant string to be a fixed length memory copy (*memcpy*), but other than that, GCC was in general a better compiler. When the same optimization was added to GCC, it got much better numbers.

There are two take-aways from this story. The first is the importance of educating your management chain so that they can champion the support of GCC. If you can't present a compelling case for supporting GCC, then the corporation might not support GCC. The second is not to rest on your past accomplishments, but you need to figure out what is needed (a new feature needed by the customer, faster benchmarks, etc.) and then implement it in a timely fashion. In a business context, this means scoping out the work, and then doing it successfully, and keeping to the schedule that you signed up for.

5.2 Full time employee, independent contractor, or non-work toolchain support

There are three main ways that a corporation might interact with a free software developer:

- If you are a full time employee who is tasked to do some work in the free software world. It might be a small part of your overall job, or it might be your full time job. Typically this is done when the company is committed to supporting GCC in some fashion or another.
- If you are an independent contractor or work for a independent company that does work for hire for the company. Often times this is done when a company wants a specific job done, but does not want to pay for people to interact with the free software community all of the time. The assumption is the contractor does all of the interaction about getting code committed, etc. but like in any business setup, there needs to be agreements about ownership of the code, who donates it, etc.

- If you work for the company doing something not related to free software, and you work on free software on your own time. If you go this route, you probably need to get the appropriate paperwork that will allow you to contribute changes.

For the full time employees there tend to be a few different categories people fall into:

- **Lone wolf**: This is where there is a single person tagged with doing free software support. It can be important to make sure your company understands why they are doing GCC support and what benefits they expect to get.
- **Developer on a team**: For most developers, this is just like any other team job within the company, with the exception that you have to deal with some more issues about getting your code accepted.
- **Free software evangelist**: In this role, you might still be a developer, but your role is to help shape how the company deals with free software in general.
- **Manager or executive over free software developers**: In this role, you are likely to have to justify the budgets for the free software support, and make sure your employees can do the job you set out for them.

5.3 Other compilers

Most hardware companies support multiple compilers, where GCC is just one possible compiler. Sometimes it is a compiler created by the hardware company and sometimes it is a third party compiler where the hardware company works with the compiler vendors. Usually these compilers are in competition with each other for the hearts and minds of the corporation and the users. We take the view that competition is good, and that users benefit from having a choice. We might not be able to make GCC always the best choice, but we try to keep it in the ball game.

5.4 Secrets

If you have signed non-disclosure agreements with your company, you need to be careful about the leakage of information via source code in the repository, in mail messages, and on relay chat lines. As we mentioned above, you need to make sure that you need to schedule when new instruction set variants are made public before contributing the code. For example, scheduling the announcement of new instruction ISA before the closure of stage 1 of the compiler, so that the code could be contributed.

5.5 Learning to think like a corporation

- Just like the free software development process has its own ways of doing things, each corporation has its own ways of doing things. To be successful in a corporation, you need to learn how to communicate effectively in that setting, and to think in terms of risks, costs, and how what you are doing matters in the big scheme of things; or, in corporate jargon, profit-and-loss, risk-reward, and expense-to-revenue ratio. Think of a corporation as another puzzle, it is another complicated system to be reverse engineered or hacked.
- Do not approach upper level management with a general complaint. People who have been promoted to upper management are very operations-oriented. They want to know what action you want them to take, not presenting them with a general problem and hoping that they invent the solution. If a hacker does not know what the executive can or cannot do, that is okay. They would prefer a strawman proposal that they can accept, reject, or modify.
- Corporations operate on profit and loss. Find out what executive, probably a general manager, that is responsible for profit and loss in your line organization and his or her goals. Either align an existing project with those goals or convince the executive that a project will help with those goals. Regardless of departments and managers and task forces, projects that help the person responsible for profit and loss will not lack resources.
- Talk is cheap; follow the money. Executives consider the cost of a request or action. An executive stating that they like / support some project or some project is great / important incurs no cost. Strategies and reports and white papers and general platitudes carry no cost. The primary question of importance is a request for resources. To understand

what is important to an executive or organization or company, look at how and where resources are deployed.

- If an executive directs the hacker to the corporate lawyer or directs the hacker to assist the corporate lawyer with risk analysis, that is a brush off and busy work. There always will be legal risk in any action. Deferring to the lawyer due to risk is a polite cop-out. The solution is not to try to resolve the legal concern, which is impossible. Instead, one must find the business justification (corporation will increase revenues X% or will lose an important customer bid). With the right business justification, the executive will decide the benefit outweighs the risk, accept the risk, and, if necessary, direct the corporate lawyer to figure out how to mitigate the risk. That is the preferred solution and the way to think like a corporation to get things done.

5.6 Maintainers

As we mentioned above, there are different classes of maintainers. If you are going to be a maintainer, be sure to get the support of the corporation in terms of the extra work that being a maintainer mandates. Being a maintainer means that you don't have to submit your patches for review as long as they are confined to the area that you are the maintainer of, but you do have to spend time reviewing patches and offering suggestions on how to approve the patches. Because maintainers / gatekeepers are a unique, limited resource, they are valued by corporations that understand free software development.

GCC is moving to a model of Reviewers instead of Maintainers. Reviewers may approve other people's patches, but not their own.

5.7 Pick your battles

Corporations each have their own little quirks, and each person has to decide what things that are important to stand up for and what things it is better to say whatever, and move on. One either can try to change the system or learn to work within the system — hack the system. In many companies, trying to change corporate policy is like tilting at windmills in Don Quixote.

5.8 Educating your management

When our management changes, we often times find it takes some time to bring the manager up to speed of the ins and outs of contributing code to GNU. This paper is an attempt to record these thoughts for use in the future.

5.9 Out of sight, out of mind

In some companies, there is only one developer who is hired to do the GNU work, and the developer is off in their own little world, while the other developers are working on the main product. It can be lonely if there is nobody else doing similar work. How you deal with the issue is up to you. Strategies include, hanging out on the IRC channels, striking up work relationships with other developers even though you don't work on the same things, or finding solace outside of your job.

Unfortunately, in these times if you are seen as disjoint to the rest of the company, it may make you the target of the next round of layoffs. Particularly if you are working at a remote location or from your home, you need to make sure that you are providing value to the company and that mangement knows what you are doing. You always need to keep in mind how what you are doing ultimately furthers a corporation's goals. If layoff time comes and your boss doesn't have an immediate answer for what have you done recently that will affect the company's bottom line, then it may be time to update your resume.

Similarly if you are higher up in the management food chain or you are a single developer in a small company, you probably need to keep a current business case of why supporting GCC is important for your company, and why they should use you to do the job instead of expecting somebody else to do it for free, and update it regularly.

Nobody owes you a job, it is up to you to provide value to the company by the work that you do.

5.10 Monolith vs. Fiefdoms

From the outside, corporations often times look like giant monoliths, speaking with one voice. However, from the inside, you often times find that you have many little fiefdoms that are competing against each other for resources and ultimately customers of the corporation.

5.11 Burnout

Doing too many all night sessions can lead to burnout. It is important to learn to deal with this before you do something that endangers your job. Part of dealing with burnout is maintaining the motivation for the job. Having outside interests can also help maintain your spirits at work. Finally, it is important in making schedules that you are realistic about the amount of time it is going to take you to do the job so that you do not have to do the all nighter because you underestimated the amount of work. If you do have long sessions where you are making a release, be sure to plan some downtime afterwards to allow you time to catch your breath.

5.12 Career Advancement

For many people, working on Gnu stuff can reduce the opportunities for career advancement, just because there are so few slots available strictly in terms of working on the toolchain. For a lot of developers that we know, this isn't a problem, in that they are happy where they are, and don't have loftier ambitions. If you do want to expand your role, often times you need to be creative and engineer the job you want at the next level.

In some companies, the usual career advancement is through the management ranks. People management is a different skill set than code development, and you need to judge whether you have that skillset or can grow into it. Not every developer wants to give up code development for writing presentations, working on spreadsheets, and managing people full time.

6 Summary

Some final thoughts:

- Make sure the legal hurdles are taken care of.
- Interact with other developers to form a personal relationship.
- Keep things at a professional level.
- Pay attention to the details and the correct procedures.
- Keep things up to date, and watch for new failures.
- Try to make smaller code submissions rather than giant all-in-one submissions.