Hunting Regressions in GCC and the Linux Kernel

Janis Johnson, James Kenefick, Paul Larson
IBM Linux Technology Center
janis187@us.ibm.com, jkenefic@us.ibm.com, pl@us.ibm.com

January 2004

1 Introduction

A software regression is a bug that exists in some version of software and did not exist in a previous version of that software. Regression testing helps to detect bugs which creep into software with new changes. In most projects, however, test suites don't catch all regressions, and some are discovered by users upon upgrading to a new release. Knowing which change introduced a regression can be valuable information for the developer who is fixing the bug. For many projects it's possible to automate searches for the causes of regressions, allowing searches to be done with less effort and higher reliability. This paper describes our experiences automating regression hunts for the GCC and Linux kernel projects, along with introductions to those projects and reasons why it's important in distributed Free Software projects to make it easier for experienced developers to fix bugs.

2 General strategy

The general strategy for identifying the cause of a regression is the same regardless of the type of software or the source control system used for it. A regression hunt uses a binary search by changeset identifier, if the source control system supports that, or by date. The first step is to identify the range of changesets or dates to search, from the latest one at which the test case is known to pass to the earliest one at which it is known to fail. The next step is to verify that the test case gets the expected result for both of those endpoints when the product is built and run using the procedures which will be used for the hunt. Starting with the original range of dates or changesets, the midpoint of the range is tested; depending on the result of the test, that version becomes one of the new endpoints of the range. This process is repeated until a single changeset or a small time interval is identified. Each check of a date or changeset updates the source code, builds the software, and runs the test case.

Folk wisdom says that when asked to identify the most powerful force in the universe, Albert Einstein replied "compound interest." Binary searches are similarly powerful. To verify that a test result changes in one five-minute period requires a check at each of the two endpoints of the range. Finding the five-minute period within a ten-minute range requires one additional check at the midpoint of the range. When the initial range is 24 hours, eleven checks are needed, including two for the endpoints. For an initial period of a year, the total number of checks is 19, fewer than twice as many as for a single day.

The example in Figure 1 shows the progress of the binary search for a regression. The first two tests are for the endpoints of the range; each further test is for the midpoint of the current range, with the version just tested becoming either the low or high point of the next range.

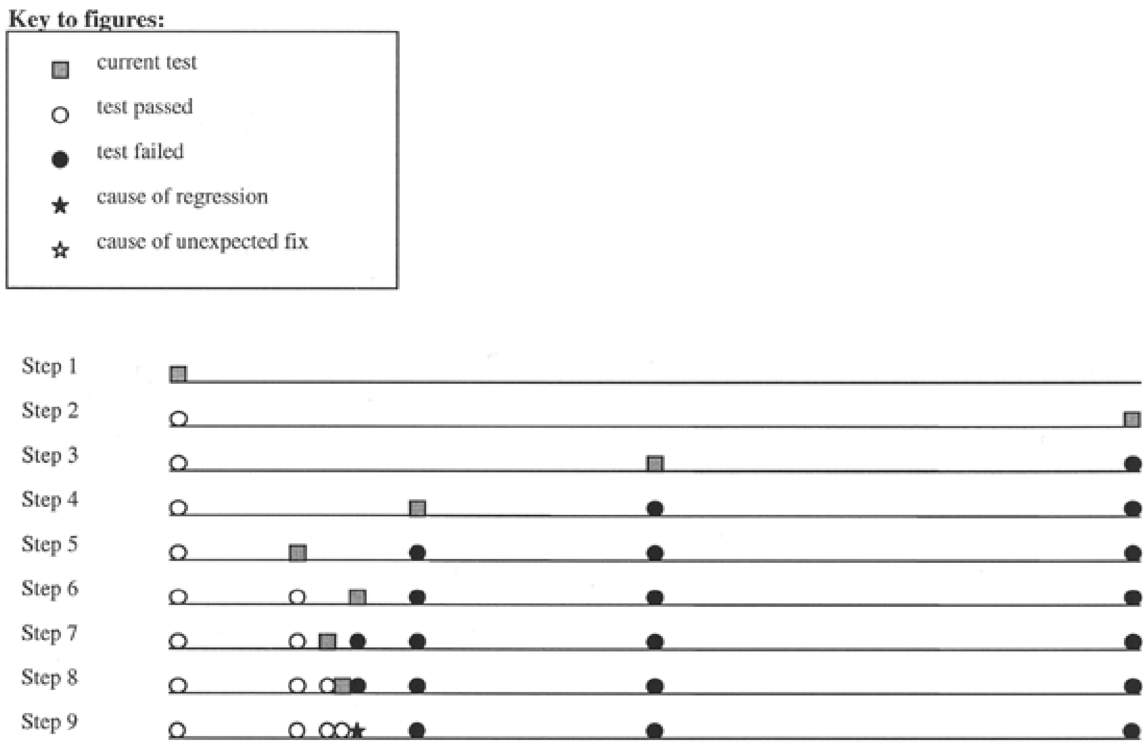

Figure 1

Checking each patch is not practical but, for illustration purposes, if each patch were tested to find the regression whose search is shown above, the results would look like Figure 2:

Figure 2

A search for the changeset or time interval for which the behavior of a test changes isn't finished until it reaches two adjacent changesets or times with different test results. A search using changesets is more balanced than is a search by date, particularly if changesets tend to be added in clumps.

The same strategy can identify the fix for a bug that unexpectedly went away. If a bug exists in one version of software but not in a similar version, such as a different branch or development tree, then a hunt can identify the fix by reversing the return code returned by the test case. If the code that was modified is similar in both versions of the software and the identified patch doesn't merely paper over the problem, then it can be ported to the version in which the bug still exists.

The decision about which range to test is more difficult for a centralized project which uses branches when searching for the cause of a bug that exists on one branch and does not exist on another. In Figure 3, the bug exists on a release branch but not on the mainline, and there is a version with known good test results on the mainline before the branchpoint:

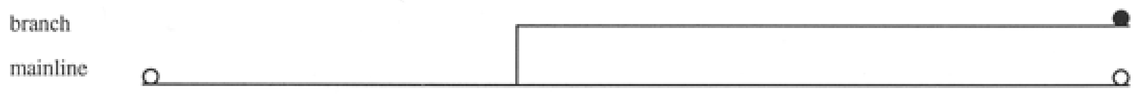

Figure 3

Testing all changesets for both branches might show that the regression was introduced only on the branch, as shown in Figure 4:

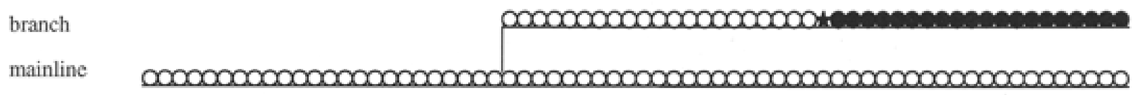

Figure 4

For another bug whose initial information was identical, testing all changesets for both branches might show that the regression was introduced on the mainline before the branchpoint, and the bug was then fixed on the mainline but not on the release branch, as shown in Figure 5:

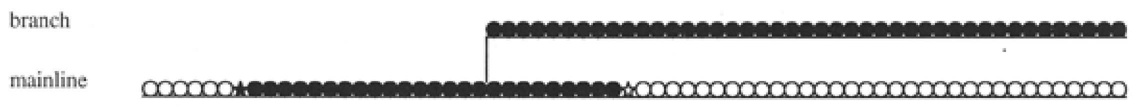

Figure 5

The strategy in this case, then, is to test the mainline just before the branchpoint to determine whether to perform a full search on the branch for a regression, or on the mainline for both a regression (between the lowest date for which the test is known to pass and the branchpoint) and an unexpected fix (between the branchpoint and a date known to have the fix).

Development branches are split off from the mainline sources and are later merged back in. A regression hunt on the mainline that identifies the merge as the cause of the regression can be followed by a new search on the development branch to determine which patch to that branch caused the regression, as in Figure 6:

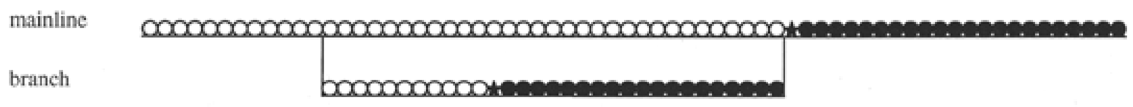

Figure 6

3 Automating regression hunts

3.1 Requirements for automating a regression hunt

Basic requirements for automating a regression search involve the source control system, the product build, and the test case.

The source control tools must allow access to all source files as they were on a particular date or when a particular changeset was added. If the project uses multiple branches of the sources, then searching for a regression on a branch requires accessing the sources on that branch as of a particular date or changeset addition. This is easier if the source control system operates on changesets rather than changes to individual files, but it's sufficient to be able to determine which files were modified for the same change even if that information is not built into the tools. The search for the cause of a regression involves accessing many sets of sources, so it must be possible to obtain each version without adversely affecting the productivity of other people who use the same source repository.

Automated regression hunts are possible only if the product build is automated and can be performed in a reasonable amount of time. For a particular regression, a hunt is possible only if there is a test case that fails when the bug is present and passes when it is not. Additionally, that test case must be automated, self-contained, reasonably quick, and safe to run so that it won't corrupt the test system. A regression hunt also requires access to a test system on which the failure can be reproduced.

3.2 Short cuts

Short cuts are often available to reduce the time needed for a regression hunt. When the remaining interval is small it's often possible to look through descriptions of the changesets that were added during that period and identify the one most likely to have caused the regression, and then test the software just before and just after that changeset was added. If the guess was wrong, the hunt can continue with that date or changeset as one of the endpoints.

A hunt based on dates can recognize when there have been no changes to the sources between the new date and one of the current endpoints of the range, and then skip the test for that date and simply make it one of the endpoints.

For some source control systems it's possible to make a local copy of the repository to avoid or lessen network overhead and to leave out portions that are not needed for the hunt, further speeding up the process of obtaining a particular version of the sources.

The build of the product need not be a complete build; it is sufficient to produce only the components that are needed to run the test. Depending on the build process it might be possible to use incremental builds rather than starting from scratch each time when the remaining search range is small and the current build uses a later version of sources than the previous build.

Some short cuts are successful most of the time but fail occasionally, such as incremental builds if not all dependencies are identified correctly. A hunt can try a short cut and then fall back to the normal procedure if the short cut fails.

If the endpoints of the initial range are not known, it's better to start with a range that is too large rather than one that is as small as possible, since an incorrect guess at a small range requires additional work by the regression hunter, while doubling the initial range requires only one additional check.

3.3 Benefits of automation

After the steps of obtaining a specific set of source code, building a subset of the product, and running a test are automated, the tools to perform those steps can be used in a fully automated regression hunt. The first few hunts generally require adjustments to those tools, but once they are stable the incremental time required to set up additional hunts is fairly low. The actual work of an automated regression hunt consists of setting up the test case and a script to build the right subset of the project, selecting an initial range of dates or changesets, verifying that the hunt reports the expected results for those initial dates, monitoring results occasionally to detect problems that require manual intervention, and then examining and reporting the results at the end.

Performing a regression hunt manually is error-prone. If there is a time lag between steps, the person performing the hunt can be distracted by other tasks and get confused about what is being tested in each step. An automated hunt whose tools create log files which record what is done at each step makes it easier to check back on what was done and recognize mistakes, as well as reduce opportunities for careless human error during the hunt itself.

Personal experience has shown that incorrectly identifying the cause of a regression can be embarrassing, and might even lead to bad haiku:

> Where's the regression?
> I thought it was located;
> alas, I was wrong.

3.4 Manual intervention

Several problems that require manual intervention can arise during an automated regression hunt. Mistakes in setting up configuration files or test scripts can cause test results to be reported incorrectly, which makes it doubly important to begin the hunt by verifying the results for the original endpoints of the range, including examination of test output to verify that a test failed for the expected reason and not because the test was set up incorrectly. There might be dates or changesets for which the product doesn't build; this is frequently the case in the GCC and Linux kernel projects, both of which support a wide variety of configurations, not all of which are tested frequently. Sometimes the test output changes due to modifications that have nothing to do with the bug, such as changes to messages that the test script examines. The build process might have changed over time, requiring different scripts to perform the build for different versions of the source. Tests for an operating system kernel can encounter a wide variety of unexpected problems.

3.5 Verification of results and intermittent failures

If an automated regression hunt uses any kind of short cuts or makes assumptions that are not always true, then it's a good idea to verify that the bug does not yet exist just before the identified changeset and that it does exist after that changeset was added. It's important to examine the identified changeset to see if it makes sense for it to have caused the bug; if not, the bug might be intermittent. A binary search for 1 or 0 will always converge, whether the results make sense or not. Figure 7 shows what it would look like to test all changesets in a range which contains a normal regression, followed by what it might look like to test all changesets in a similar range which contains an intermittent bug. The checks and results for a binary search which could match either of those patterns (this is the same binary search which was illustrated earlier).

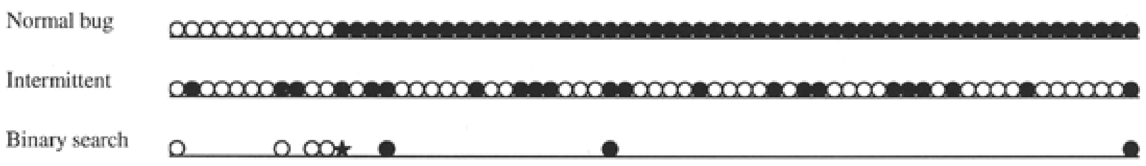

Figure 7

Testing at regular intervals can show whether a failure is intermittent, depending on how often it fails and whether the periodic testing is lucky and shows the irregularity of the results. In Figure 8, the first set of periodic checks uses a large interval that happens to show the same pattern as would be seen for a regression, while the two sets of checks with smaller intervals demonstrate the intermittent failures:

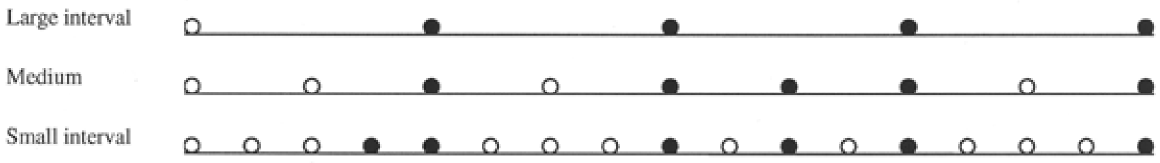

Figure 8

Some tests might get different results when run multiple times in the same environment. For those tests, a check of a particular version can run the test multiple times and report it as failing if any of the runs fail.

4 Hunting regressions in GCC

4.1 The GCC project

The work described here was first performed within the GCC (GNU Compiler Collection) project. GCC is part of the GNU Project, which is Free Software covered by the GNU General Public License, with copyrights held by the Free Software Foundation [1]. GCC includes compilers for C, C++, Fortran 77, Java(TM), Objective-C, and Ada, and supports a wide variety of processors and several operating systems, including embedded systems. GCC is the primary compiler used for GNU/Linux and other Free Software and open source operating systems and is the system compiler for Mac OS X. As an example of the variety of targets GCC supports, in the first two weeks after the release of GCC 3.3 there were reports of successful native builds for 27 unique targets, with a combination of 16 processors and 10 operating systems [2].

GCC is developed and maintained by a large number of contributors who are either volunteers performing the work on their own time or people whose work on GCC is funded by their employers or through contracts. Twelve global maintainers can make or approve changes to any part of the project, a few dozen additional maintainers can make or approve changes to specific parts of the project, and another hundred or so people have write access to the sources, allowing them to check in their changes after approval by a maintainer. Many other people contribute but are not able to check in their patches themselves. This development model can be very surprising to people who have not been exposed to it, but it has been successful for many Free Software and open source projects [3].

GCC contributors tend to focus primarily on current development; volunteers work in areas that are of personal interest to them, and funded contributors do work that is a priority to their employers. Stabilizing an upcoming release receives adequate attention, but fixing regressions for bug-fix releases is not a particularly popular activity. GCC's Release Manager can encourage contributors to fix bugs but has no real authority over them; from his perspective they are all volunteers.

GCC is a large, complex project [4], and many GCC bugs can only be fixed by an expert who has a deep understanding of the interactions between the various parts of the compiler, which is usually gained through of years of experience. Other contributors who want to help get bugs fixed can often provide the most support by providing information and analysis that will make it easier for the experts to fix bugs.

4.2 GCC processes

4.2.1 Source control

The GCC project uses centralized source control using CVS [5]. CVS handles changes to individual files rather than changesets that modify a set of files. A CVS commit of several files is not atomic, although it does support providing a list of files that are usually checked in within a very short period of time. By convention, each GCC patch is described in a ChangeLog entry that lists all of the files that are modified for that patch; the same information is included in the CVS log of each of those files. These conventions allow a human being to recognize changesets even though automated tools only know about changes to individual files.

CVS supports multiple branches in a single source tree, and the GCC source tree includes several branches. The mainline development branch is used for new functionality that is ready for use by the GCC development community. A major release gets a branch when all new functionality for that release is in the mainline and is relatively stable; that branch continues to be used for later bug-fix releases. Experimental work is often performed on a development branch which is merged back into the mainline when it is stable. CVS tools support automated merges of mainline changes into a branch, and GCC development branches generally get regular merges from the mainline.

Figure 9 shows the mainline, release branches, and representative development branches of the FSF GCC CVS tree between January 2001 and June 2003. The branches shown above mainline are release branches, with '*' showing a release. The branch for 3.2 releases is unusual in that it is actually a continuation of the 3.1 branch. Branches shown below mainline are development branches that were merged back into mainline. During this time there were several additional development branches that are not shown here.

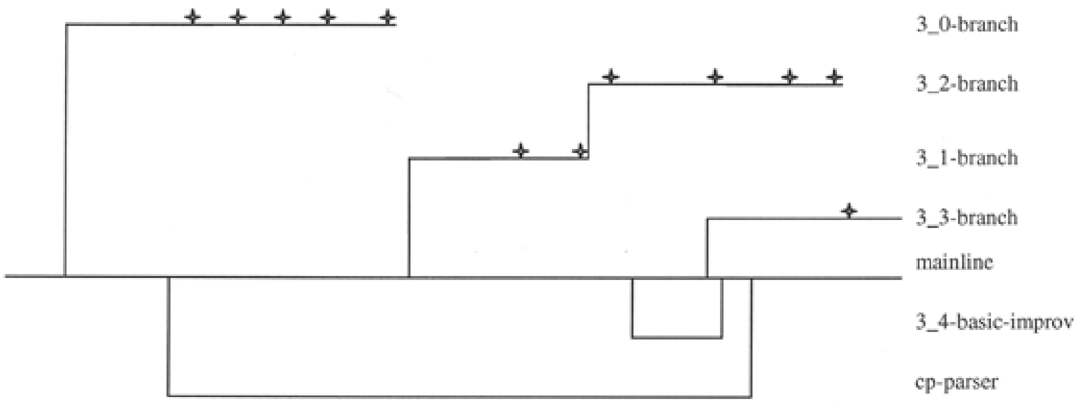

Figure 9

4.2.2 Development process and regressions

The GCC development process [6] specifies that bug-fix releases only fix regressions. This makes it important to identify which bugs are regressions so that they will be fixed in a bug-fix release.

When a patch to the mainline causes a significant regression, such as a build failure on a major platform, that patch must be reverted if the regression is not fixed within 48 hours. Often the cause of a new failure is not obvious and must be hunted down.

4.2.3 Bug tracking

GCC's bug tracking system is available for anyone to view via the GCC home page, and anyone can submit a problem report against the FSF GCC sources. The instructions for reporting a bug [7] specify that a problem report must be accompanied by a self-contained test case and enough information about the software environment to allow GCC contributors to reproduce the problem.

Problem reports are analyzed by volunteers. Currently there is a dedicated group of volunteer "bug masters" who do this very quickly. A volunteer determines whether a bug report is valid (is not based on a misinterpretation of the rules of the language or some other user error). He or she then determines whether the bug still exists on active branches and whether it is a regression from a released version of GCC. Some of these volunteers provide minimized versions of the test cases that are submitted with the problem reports; it's not unusual for a talented bug minimizer to cut down a 30,000-line test case to 10 or fewer lines of code that trigger the same compiler bug. In addition to being extremely valuable to the people who fix bugs, these minimized test cases can be used in regression hunts.

4.3 Types of compiler bugs

Compiler bugs come in a wide assortment of flavors: incorrect code generation causing the compiled test case to abort or to produce incorrect output; internal compiler errors for valid or invalid source code input; an undetected error or missing warning, or an incorrect error or warning message. Each of these requires a source file to compile, perhaps a specific set of compiler options, and a test script which recognizes whether or not the test case was compiled correctly.

4.4 Tools for hunting regressions in GCC

GCC contributors have been identifying the causes of regressions for a long time, but until recently the methodology was not described and there was uncertainty about whether it was even possible to obtain the sources from a CVS branch for a particular date. In December 2002 there was a big push among the "bug masters" to minimize test cases, determine which bugs were regressions, and in general provide as much useful information as possible to the people with enough experience to actually fix the bugs. There was a concerted effort at that time to identify the causes of regressions, and the people involved agreed to share tips and to document them [8]. During this effort it became apparent that it is possible to automate regression hunts.

A shell script called `reg_search` was developed at that time to provide the framework for automated regression hunts. Most GCC developers prefer to use their own scripts to obtain sources and to build GCC, so `reg_search` invokes separate tools to perform those functions. The various kinds of compiler bugs require different kinds of tests, so the script that runs a particular test merely returns a value of 1 if the search should continue with later dates, or a value of 0 to search earlier dates. A configuration file specifies the locations of these scripts, plus the low and high dates of the range to search and the desired length of the final period to report (5 minutes by default).

GCC runs on systems where other GNU tools are either available by default or are easy to build and install, so `reg_search` takes advantage of a nonstandard extension to GNU `date` to format dates as the number of seconds since 1970-01-01 00:00:00 UTC [9]. `reg_search` converts all dates to this format, making binary searches by date straightforward.

`reg_search` supports continuing a search after manual intervention has been required. When a source update or build fails, `reg_search` reports the low and high dates of the range with which the search should continue. If GCC fails to build for the tested date, the configuration file can specify an initial midpoint to test rather than using the actual midpoint of the current date range. By default, `reg_search` begins by verifying the initial low and high dates, but these checks can be skipped if they've already been verified by hand or in an earlier stage of the search.

A related shell script, `reg_periodic`, builds and tests at regular intervals in a specified range of dates. This is useful when the regression hunter suspects that a bug might be intermittent, to find out whether the test result varies over time. Running a test at regular intervals is also useful when hunting for a performance regression for which there might be several small performance degradations rather than a single large one.

`reg_search` and `reg_periodic` are available in the GCC source tree in a directory of contributed tools [10]. They are copyrighted by the FSF and licensed under the GNU General Public License. Besides depending on an extension to GNU `date`, they use features of `bash` (GNU Bourne-Again SHell) [11] that are not available in other shells. These scripts contain no assumptions that they are being used for the GCC project or for a compiler; they can be used for any project which requires searches by date rather than by changeset identifier.

4.5 Benefits to GCC

Between December 2002 and April 2003 there were 32 regressions which affected the then-upcoming GCC 3.3 release whose causes were identified and the bugs fixed. During the same period, 85 other regressions were fixed for the GCC 3.3 branch. Most of these were regressions which were introduced during the development of that branch, but some also affected released versions of GCC. One might have expected that the regressions whose causes were identified were fixed sooner than the others, but that didn't turn out to be the case. The people who fixed those bugs, however, agreed that in general knowing the cause of the regression was valuable, although having a minimized test case is even more helpful than knowing the cause of the regression. None of these people had fixed enough regressions with identified causes to be able to quantify the time savings.

The value of knowing which patch introduced a regression varies widely depending on several factors, including the nature of the bug, the size of the patch, the familiarity of the bug fixer with the modified code, and whether the patch introduced a new bug or merely exposed an existing latent bug. In some cases, a close examination of the identified patch reveals a mistake or omission in logic allowing the bug to be fixed easily. If the identified patch was a large change for new functionality or a rewrite of a portion of the compiler, at least the fixer knows in which part of the compiler a bug might lie. Identification of a patch that exposed a latent bug allows examining debugging dumps of compilers with and without the change, or setting up simultaneous debugging sesssions of those two versions of the compiler, to determine at which point the behavior changed.

Another benefit of identifying the cause of a regression in a volunteer project like GCC is that regular contributors are more likely to look at problem reports for bugs which they have introduced. In some cases that's when an expert notices that the code in the test case is ill-formed, which is particularly common for inline assembly code and for C++ code which the compiler has rejected. GCC's Release Manager is able to use the information about which patch introduced a regression to ask the author of that patch to investigate and fix the problem.

5 Hunting regressions in the Linux kernel

5.1 Overview of the Linux kernel project

Like the GNU Compiler Collection, the Linux kernel is Free Software distributed under the GNU General Public License. The Linux kernel supports a wide variety of hardware platforms and CPU architectures and handles everything from low-end desktop systems and embedded systems to large multiprocessor enterprise-class systems.

As with GCC, Linux kernel development is performed by programmers all over the world. Main versions of the kernel have maintainers, as do specific subsystems and drivers. Any developer can contribute code, from a simple fix to a large feature, by submitting it to the Linux kernel mailing list. Many patches are accepted into a maintainer's tree, or into a patch set outside the main kernel, before becoming part of an official tree.

The Linux kernel has both stable and development versions. A kernel version with an even number (including 0) in the second position is considered stable, while a version with an odd number in the second position is a development version. The 2.5.x versions of the Linux kernel were started shortly after the release of 2.4.0 and have since been developed in parallel with 2.4. When the 2.5 kernel is complete it will become 2.6, indicating that it is now a stable version. For the most part, only bugfixes, drivers, and very small but important features are added to stable trees. Development trees receive much larger and more experimental changes.

5.2 Linux kernel development processes

5.2.1 Source control

Linux kernel source trees are maintained using a variety of formats and tools, but primarily BitKeeper or combinations of tarballs, patch sets, and collections of individual patches.

Linus Torvalds uses the revision control system BitKeeper [12] to maintain the official Linux source trees. Many other Linux maintainers and developers also use BitKeeper, which was designed for the development model used by the Linux community [13]. BitKeeper tracks the revision history of a source tree at the changeset level, where a changeset is one or more changes made at the same time to one or more files. Anyone with access to the BitKeeper tools can clone any public BitKeeper tree, create a snapshot of the tree, and generate GNU-style patches for any changeset in the tree.

The 2.4 and 2.6 Linux kernel trees are available as a read-only CVS repository. The '`bkcvs`' repository is generated from BitKeeper trees and contains much of the changeset information that BitKeeper records. BitMover generates the '`bkcvs`' repository as a service to the Linux community using an internal tool which is heavily dependent on BitKeeper internals.

Linux kernel sources are released as gzipped tar files available via `ftp`, with diffs available for all changes between releases. Individual patchsets are also available as GNU-style patches, generated by BitKeeper, for recent releases.

5.2.2 Bug tracking

Linux kernel bugs are generally reported on the Linux kernel mailing list. Anyone reading that list is free to submit a patch which fixes a bug. Some bugs that are reported this way are forgotten unless their submitters are persistent. Therefore, the report of a regression can go unnoticed for quite some time, making regression hunting more important when the bug is eventually noticed again.

A bug tracking system based on Bugzilla [14] is also used to track Linux kernel bugs. Some people still prefer the mailing list approach, but many are beginning to recognize the benefits of a formal bug tracking system.

5.3 Types of kernel bugs

Kernel defects can generally be classified as either build or operational; they manifest themselves either during the build process or while the kernel is in operation. Build defects prevent successful compilation of the kernel. They are good candidates for automated regression hunting, where the test driving the binary search is the kernel build itself.

Operational defects come in many forms. The simplest type to identify with a regression hunt is one that is quickly and easily reproducible and non-fatal. Examples include a system call that returns an invalid error code, functionality that fails but doesn't report an error, and a non-fatal kernel oops. This type of defect can generally be reproduced with an automated test in a short amount of time and does not cause a fatal system error such as a kernel panic. Defects that don't meet these criteria are generally not good candidates for automated regression hunts. Race conditions, in particular, often take a large or unknown amount of time before they can be reproduced. Defects that cause a fatal error require manual intervention to continue a regression hunt, as do those that halt the boot process or cause an interruption in network communications. Even bugs like these, however, can be worth identifying with a regression hunt if they are particularly difficult to understand.

5.4 Tools for hunting kernel regressions

5.5 Framework

Automated regression hunts for the Linux kernel are based on patchsets rather than dates, so a new version of `reg_search` uses integer identifiers to control the binary search. The tools described below are not yet in real use, but they are a proof of the concept and are being considered for use in test systems within the IBM Linux Technology Center.

5.5.1 Source updates

BitKeeper supports, via the `bk export` command, accessing sources as they were for any release, as of any particular date, or at the time any changeset was added. A regression hunt with a BitKeeper tree is most efficient using a clone of the tree on the system where the sources are needed. The `bk prs` command can extract information for each changeset including a revision identifier, date, and symbolic tag (if any). The regression hunt tools assign an index to each of the changesets in the range of interest and use that index to control the binary search.

The '`bkcvs`' repository is also well suited to regression hunts. The CVS log for an empty file called '`ChangeSet`' includes, for each changeset, that changeset's unique BitKeeper identifier, the date it was added to the BitKeeper tree, and a logical change identifier which is recorded in the CVS log for every file in the changeset. Unlike a normal CVS tree, every file in a changeset has exactly the same checkin date. It's a simple matter to map between the '`bkcvs`' changeset identifier, the checkin date, and the unique BitKeeper identifier, and to map from the symbolic tag for a release to a changeset identifier. The regression hunt framework performs a search based on changeset identifiers, which are mapped to dates for use with CVS operations.

Drawbacks of using the '`bkcvs`' repository are that because BitMover provides this as a free service to the Linux community, it makes no guarantee that the gateway will continue to be available or supported, or that the amount or format of information in the CVS logs will remain the same. The gateway is only available for the 2.4 and 2.6 integration trees. BitKeeper and CVS were designed for very different development models, and not all of BitKeeper's information can be translated to CVS.

It should be possible to perform automated regression hunts based on snapshots and patches that are applied to them, but the authors haven't yet tried this approach.

5.5.2 Build, boot, and test

For most types of kernel regressions, it's easiest to use two systems for a regression hunt: one system to control the regression hunt, obtain the sources, and build the kernel, and a second system on which to boot the kernel and run the test.

The target system is chosen and the kernel is configured and built to allow the defect to be reproduced for versions that have the bug. In many cases the configuration can be very simple, with any necessary options built into the kernel rather than using modules.

The test system is set up to boot a new test kernel by default, with each new version of the test kernel having the same name. The automated hunt copies the new kernel to the test system, updates the bootloader if necessary, performs any other required steps, reboots the system remotely, and verifies that the boot was successful.

The final step of testing a particular version of kernel sources is executing the regression test. The test files are placed on the target system ahead of time and run remotely for each check, with information available to the regression hunt tools about how to run the test. The result returned to `reg_search` is a value telling it whether to continue the search with later or earlier versions of the sources.

5.5.3 Recovering from build failures

Build failures for GCC, while not unusual, are not common enough to require avoiding manual intervention. The version of `reg_search` developed for use with GCC provides a mechanism to skip the current endpoints of the range if they have already been tested and to specify a particular date with which to start a new round of testing. This is sufficient for the frequency with which which GCC builds fail.

On the other hand, build failures are quite common with the Linux kernel. The adaptation of `reg_search` used for the kernel has an optional callout which takes the endpoints of a range in which a build has failed and returns the identifier of a different changeset to test next. Keeping track of failing builds allows for trying several different changesets after a failure before continuing the hunt with a successful build. Even heuristics that are not very efficient are worthwhile if they can avoid manual intervention.

5.5.4 Identifying the introduction and fix of a build failure

When there are several build failures for the same reason that appear to be within a continuous range, automated regression hunts can identify both the introduction of the build failure and the changeset that fixed that failure. While hunts for most bugs are most easily done using one machine for the build and to run the hunt and a separate system on which to boot the new kernel and run the test, the hunt for a build failure needs only a build machine.

When searching for a build failure, the test step is the build itself. The build step can be replaced with `true`, and the test step consists of performing the build and, if it fails, comparing the failure (as shown, for example, with ‘`grep '^make.* Error'`’ on the `make` output) with similar output for the expected failure.

In many cases it's possible to extract into a new patch those parts of the changeset which allow the build to succeed, and to apply that patch to each version of the sources within the range of that failure. This makes it possible for other regression hunts to search within that failure range.

5.6 Benefits to kernel developers

The benefits of automated regression hunts in the Linux kernel are expected to be much the same as the benefits of using them with GCC. With BitKeeper used for revision control, a detailed list of changesets is tracked for every new kernel version. Typically, even large changes are broken up as much as possible into multiple smaller changesets. Before all of these individual changesets were tracked in the official Linux kernel, the best anyone could hope to do was figure out in which version of the kernel a regression first appeared, which wasn't very helpful when each version can include hundreds or thousands of changes. Tracking for each changeset, combined with the ability to automate the process of searching them, can narrow the cause of a regression to a single changeset, which in many cases contains as little as a line or two that changed. Armed with this knowledge, a tester can often identify the specific cause of a bug, and sometimes even fix it without requiring the original developer to spend time tracking down where the bug exists, or even whether it exists within their component of the kernel.

6 Conclusion

Automated regression hunts are, in theory, possible for many projects, and their results are less prone to error than are the results of manual regression hunts. The value of identifying the patch which introduced a regression varies widely, but for the GCC project there have been enough cases where the information is useful to make regression hunts worthwhile.

7 Acknowledgements

This paper is based on the authors' experiences hunting for the causes of regressions in GCC and the Linux kernel and developing `reg_search`, `reg_periodic`, and other tools for use with those projects. Thanks again to GCC contributors Wolfgang Bangerth and Volker Reichelt for their help with GCC regression hunts and to Richard Henderson, Jason Merrill, Mark Mitchell, Nathan Sidwell, and Zachary Weinberg for their feedback about regression hunts in GCC. Dave Hansen patiently answered lots of questions about Linux kernel source control practices. Larry McVoy provided valuable information about BitKeeper and the BK—>CVS gateway. Dorothy Bayern once again provided valuable review comments.

Legal Statement

Much of this paper was included in *Automating the Hunt for a Software Regression* by Janis Johnson, presented at the Pacific Northwest Software Quality Conference (PNSQC) in Portland, Oregon on October 13 - 15, 2003.

References

[1] Richard Stallman, *The GNU Project*,
http://www.gnu.org/gnu/thegnuproject.html .

[2] Build status for GCC 3.3,
http://gcc.gnu.org/gcc-3.3/buildstat.html .

[3] Eric S. Raymond, *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*,
O'Reilly, 1999.

[4] Zachary Weinberg, *A Maintenance Programmer's View of GCC*,
Proceedings of the GCC Developers' Summit, 2003,
http://wwwlinux.org.uk/~ajh/gcc/gccsummit-2003-proceedings.pdf .

[5] Per Cederqvist et al, *Version Management with CVS*,
http://www.cvshome.org/docs/manual/index.html .

[6] GCC Development Plan, http://gcc.gnu.org/develop.html .

[7] GCC Bugs: Reporting Bugs, http://gcc.gnu.org/bugs.html .

[8] How to Locate GCC Regressions, http://gcc.gnu.org/bugs/reghunt.html .

[9] GNU Core Utilities, http://www.gnu.org/software/coreutils/ .

[10]
http://savannah.gnu.org/cgi-bin/viewcvs/gcc/gcc/contrib/reghunt/#dirlist .

[11] Free Software Foundation, *Bash Reference Manual*,
http://www.gnu.org/manual/bash/index.html .

[12] BitKeeper FAQ,
http://www.bitkeeper.com/Documentation.FAQS.Linux.html .

[13] Val Henson, Jeff Garzik, *BitKeeper for Kernel Developers*,
http://www.nmt.edu/~val/osl/bk.pdf .

[14] Kernel Bug Tracker, http://bugme.osdl.org .