Wine Code Coverage Analysis
Midyear Report

Dan Kegel
Google Liaison

Maleina Bidek
Google Liaison

Elizabeth Sweedyk
Team Advisor

Aaron Arvey
Claremont McKenna

Edward Kim
Harvey Mudd

Evan Parry
Harvey Mudd

Cal Pierog
Harvey Mudd

December 8, 2004

Abstract

This project will help ensure that Google's Windows applications run properly on Linux. We have improved coverage tools to identify areas of untested code used by an application. We will now use these new tools to identify bugs in Wine that affect Google applications, focusing specifically on the Picasa application.

Contents

1 Background

During the last decade, the Internet and personal computing have grown from luxuries to household necessities. At the same time, Microsoft has enjoyed a monopoly on the personal computer operating system market. Essentially all PCs sold today in the US are bundled with Microsoft Windows, and a large percentage of those bundles also include Microsoft Office. Large PC vendors such as Dell receive 'marketing incentive' payments from Microsoft to ensure that this situation continues. Free of any meaningful competition, Microsoft has essentially been free to dictate the price it charges for its software, and as a result is one of the most profitable companies in the world.

During the same decade, Linux has matured as a server operating system, and is now beginning to mature as a desktop operating system as well. Three examples of high quality Free software often bundled with Linux include the Mozilla web browser, the OpenOffice.org office suite, and the Gnome desktop. All three of these projects are of increasing interest to the large customers who pay large site licenses to Microsoft, because together they offer something novel: competition.

Recently, hardware markets have tightened considerably and prices for the actual machines have fallen dramatically. As a result, software licensing fees as a percentage of total cost have risen, causing many companies and governments to look to the free Linux software as an alternative to the costly Microsft software. Some companies like IBM, Sun, Novell, and HP are even pouring billions of dollars into enhancing Linux to suit their needs. The IDC is predicting that Linux's share of the desktop market will rise to 7 percent by 2007.

However, there is a chicken-and-egg problem with switching to Linux: software vendors won't write Linux versions of their programs until a large portion (say, 20%) of personal computers are sold with Linux, and nobody will buy a personal computer with Linux until applications are available. (Antitrust lawyers call this the Applications Barrier to Entry; it's what prevents new operating systems from entering the market.)

The Linux community recognized this problem and began the Wine project, which aims at implementing the Windows API under Linux. In theory, Wine would allow all Windows programs to run under Linux.

In the recursive acronym tradition of open source, Wine stands for "Wine Is Not an Emulator." This refers to the fact that Wine does not try to emulate a complete hardware environment, but rather allows the Windows binaries to run directly on the CPU. Wine's job is to intercept Windows API calls and seamlessly perform the desired behavior under Linux.

1.1 Why does Google care about Wine?

Google is best known as a search engine giant, but its mission is wider: Google wants to organize the world's information, and make it easy to use. Since lots of information is stored on desktop computers, Google naturally wants to help out there, too. Its first offering in this direction is Picasa, a program to help users organize their photographs.

Google, like other software developers, is quite aware of the statistical imbalance in OS usage mentioned above and, not surprisingly, has developed its applications for Windows.

Google does not yet have sufficient incentive to port its Microsoft Windows applications to Linux, but would still like its applications to be available to users at companies that have migrated to Linux. The most economical way to do this is to run Google's Microsoft Windows applications under Linux using Wine.

1.2 Wine is not the silver bullet

Although a decade of work has been contributed, Wine still labels itself as alpha software. The main problem is that the Windows API is a moving target. The Wine project began as an attempt to run Windows 3.1 applications. Since then Microsoft has released Windows 95, 98, ME, NT 4, 2000, and XP, in addition to many new APIs such as the DirectX media libraries or the Winsock networking calls. The Wine project has made a valiant effort to keep up all the new Windows functionality while continuing to develop on the rapidly moving Linux platform.

It comes as little surprise that Wine has a rather uneven level of support for various Windows libraries. Due to the volunteer nature of open source development, the essential libraries and nifty features have generally been implemented before the boring or hard bits. In addition, the desire to make things work now has led to some neglect of documentation and testing. As things are, it is hard to say with great confidence what is well supported (and tested) under Wine, what is partially supported, and what just plain does not work.

2 Writing Tests for Wine

In order to take the place of the Windows API, Wine must catch any Windows API calls made by an application, and transparently provide the same functionality under Linux. To achieve true transparency, Wine needs to implement a function which will take the same parameters, perform the same desired functionality, and return the same results as in Windows for each API call.

We can assure ourselves of this property by writing a test which calls a particular API function. A test is essentially a small program which only calls the APIs we are trying to test in addition to a bit more supporting code for creating the desired test conditions. To actually code a test, we read the documentation and know what results we expect. Wine has specially constructed its test infrastructure to allow the tests to be compiled and executed using both Wine in Linux and the Microsoft libraries in Windows. If the Wine implementation is correct, test execution under Wine should perform exactly the same as when we use the actual Windows libraries.

2.1 Our First Test

As part of our testing of Google applications in Wine, we want to identify areas of Wine which Google applications use but are not well tested by the test suite. Areas which are "not well tested" lack sufficient or perhaps any test code. One of our main tasks after identifying these areas is to write appropriate tests when possible.

The rationale here is to determine whether the untested Wine code does in fact behave correctly. Code which does not pass tests is very likely to cause trouble for applications using that code. If we locate areas of code producing erroneous results we can notify Wine with bug reports or attempt to fix the bug ourselves and submit a patch.

With this goal in mind, one of our preliminary projects has been to develop and submit a test program for the previously untested LZExpand DLL, which provides file reading functions for accessing compressed files. This project was begun before any serious testing of Google applications as a way to learn the structure of the Wine test suite, how to write tests, and how to submit code to the Wine project.

2.2 The Test Suite Structure

Test programs are located with the code in Wine they test. They are normally put in a "tests" subdirectory of the main code directory. So for example, the LZExpand test code is located in the directory `dlls/lzexpand/tests`.

The tests are written in C and use a large supporting code framework common to all tests. The driver function is called `START_TEST(testname)`. This is called when the test is executed. The programs follow fairly normal execution from there. The test suite aggregates statistics from all the tests on all parts of Wine. Test success is reported out of the program using an assert like function called `ok()`, which takes a boolean expression indicating correct operation (eg. checking a pointer is not `NULL`), and a string which explains the problem if the check fails.

This means we need a mechanism for checking the veracity of return values and function call results. This is often accomplished by hard coding the correct result into the program and checking against that. For example, when checking that the LZExpand DLL correctly decompresses and reads a file, we will encode the decompressed file data in our program and check that against the result of the read call.

2.3 The LZExpand Test

We have written and submitted a fairly comprehensive test program for the LZExpand DLL function calls `LZOpenFile()`, `LZRead()`, and `LZCopy()`. The implementation of this test was fairly straightforward, following the conventions mentioned above. For example, for `LZOpenFile()`, we call the function before creating a file and after creating a file with different flags. Before the file has been created, we check that calls which open the file for reading correctly produce a non-existent file error. After the file is created, we check that calls to open the file for reading return a valid file handle.

The next few sections will discuss specific problems we needed to address in writing the LZExpand test.

2.3.1 Reading a Compressed File

Part of our testing required decompressing and reading a compressed file using the LZRead() call. The LZExpand DLL operates on files compressed using a variant of the Lempel-Ziff compression algorithm as produced by the compress.exe program from Microsoft. To test the reading functionality, we needed a compressed file our test program could operate on.

Since tests are discouraged from including extranenous data files beyond the source code, we elected to include a hex string in the code which represents the compressed file. When we need the file, we write it out to disk (using the normal file manipulation functions, which we assume are correct), and then use the LZ functions to manipulate it.

2.3.2 Alternate File Endings

The MSDN specification for `LZOpenFile` states: "If `LZOpenFile` is unable to open the file specified by `lpFileName`, on some versions of Windows it attempts to open a file with almost the same file name, except the last character is replaced with an underscore ("_"). Thus, if an attempt to open MyProgram.exe fails, `LZOpenFile` tries to open MyProgram.ex_."

It is unclear what the desired behavior for Wine is in this case. On some versions of Windows, if the filename does not match exactly, failure results, while on other versions, the filename with a "_" ending is tried also. Our testing has indicated that the former behavior exists on Windows 98, while the latter behavior is exhibited by Windows XP and Wine. The Wine testing guidelines say we should accept the behavior of any valid version of Windows, which means our test cannot judge one behavior or the other to be correct. With such a predicament, we have left this section of code commented out.

This case also highlights a difficulty in using MSDN as the API reference. Since it attempts to cover all of the Microsoft APIs for all versions of Windows, MSDN is not very precise or up-to-date. This is not a surprising state of affairs since computer documentation is often deficient. However, it reinforces the idea that testing under Windows and Wine is the only real way to ensure capatibility.

2.3.3 Reading Past File End

Another problem was a bit stranger and one we have not completely resolved. To rigorously test the `LZRead` behavior, we had our test read twice as much data as should have been possible from the file. From the MSDN documentation for `LZRead`, it appeared this would be handled as one would expect from the C library. That is, the read function should return the number of bytes actually read (which, in this case, would be less than the number requested), but it should not be an error. This was the functionality we observed with Wine under Linux. However, attempting to read past the end of the file in Windows produces a read error. This is in contradiction to the MSDN documentation, which our liaison warned us is likely to be wrong.

3 GCOV and LCOV

As was stated earlier, this clinic team is writing tests for the Wine project. However, another large focus of the clinic is to examine methods for writing tests. Typically a test is written either by the developer who wrote the code or sometimes in large projects, people are designated test writers. Tests are meant to break the code in any way possible through stress testing, boundary cases, unexpected input, integration testing, unit testing and many other methods. It is the hypothesis of the clinic project that for a large project already in progress, it will be easier to write directed tests if we know which code has already been tested.

This hypothesis seems natural and perhaps obvious, yet it is far from so. Code that has already been tested may have only been tested in a manner which was not strenuous enough. In addition, even if it were to be the case that there was a crash in an untested section of code, it may have very well been an unstable environment due to any number of other causes. These are just a couple of the exceptions to the hypothesis. However, many hypotheses have exceptions and ours is no different.

We will test our hypothesis by using code coverage tools such as GCOV and LCOV which will be discussed in this section. Once we know the statistics gleaned from these programs, it is our intention to write tests that cover the code, thus increasing the reliability of Wine.

3.1 GCOV

3.1.1 What Does GCOV Do?

GCOV[1], which stands for "GNU Code Coverage", is a tool developed by the GNU folks for the gcc compiler suite. GCOV is used to profile code execution statistics concerning which source lines of actually implemented machine instruction code were run. For instance, a source file `source.c` is compiled using `gcc` with the appropriate switches (`-ftest-coverage` and `-fprofile-arcs`). These switches cause gcc to output several additional files which will be used to determine which lines are run during execution. We can then run our program in whatever manner we see fit. When the program terminates, we can run `gcov` on the original source file `source.c` and an output file, `source.c.gcov`, will describe how many times each line of source code was run during execution.

3.1.2 Why We Use GCOV

We will not be using GCOV directly in our project. We will be using a wrapper around GCOV which was developed by the LTP (Linux Test Project) called LCOV. This wrapper does several things which enables GCOV to be applied to larger projects. Please see the LCOV section (3.2)to learn more about the tool and how we modified it.

[1]Documentation can be found at `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html` and the gcc compiler suite can be downloaded at `http://gcc.gnu.org/`

3.1.3 Some Internals of GCOV

GCOV needs several things in order to work. First and foremost a developer must compile the application with all the additional information needed to run GCOV. This information includes three files named *sourcefilename*.{`bb`,`bbg`,`da`}.

sourcefilename.`bb` is outputted at compile time when the `-ftest-coverage` switch is used. The `.bb` file enumerates the other source and header files which are referenced by the file that is currently be compiled. It also keeps track of the line numbers in the file being compiled.

sourcefilename.`bbg` is outputted at compile time when the `-ftest-coverage` switch is used. The `.bbg` file contains possible branches from each block of code. This allows gcov to do a full construction of a program flow graph.

sourcefilename.`da` is outputted after a program compiled with `-fprofile-arcs` is run. The `.da` file contains numbers which represent how many times a line of code was run. The numbers are cumulative, which means that if you run the program twice in the exact same manner, then all the numbers in *sourcefilename*.`da` will be doubled.

More information can be found at `http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html`. This page also describes the specific formats of the three files.

3.1.4 How to Apply GCOV to Wine

As stated before, we will not be applying GCOV directly to the Wine project. However, there is a brief tutorial in Appendix A that goes over how to use gcov on a single source file in wine. This documentation was submitted to the wine project in the patch which enabled GCOV code coverage abilities to be compiled in to Wine.

3.2 LCOV

3.2.1 History

Historically, LCOV[2], which stands for "LTP GCOV" has been used by the LTP project to see what parts of the Linux Kernel have been tested. LCOV was developed by people at IBM Linux Technology Center in Austin and the IBM Development Lab in Boeblingen, Germany. Given the Wine project is semi-similar to the an operating system environment, such as the Linux Kernel, it was thought that this tool could assist in test writing.

3.2.2 What Does LCOV Do?

LCOV, which is implemented in PERL, is a wrapper around GCOV. It automates a lot of tasks that would be very daunting otherwise, such as generating HTML output and running

[2]Further description and documentation can be found at `http://ltp.sourceforge.net/`

GCOV on a whole directory of source files. Since LCOV is just a wrapper, a developer still needs to compile the project with the project with the necessary flags as referenced in the GCOV section. Once, the project is compiled and has gone through an execution, one can do the following:

```
%> lcov -c -d build -o tracefile.info
%> genhtml -o lcov.out -s tracefile.info
%> mozilla lcov.out/index.html
```

Figure 1: LCOV example run

This will create a directory called `lcov.out` which will have a `index.html` file. This file will have statistics about the entire project, such as total number of implemented lines of code and how many of these lines were executed (figure 2), and a list of links to directories (figure 3) and files (figure 4)that are in the project. Each file will have lines beginning with numbers representing the number of times the line was executed. The lines are also highlighted according to whether they were executed, blue for executed and red for not executed, as in figure 5. This tool makes it very clear what is and is not tested in the project.

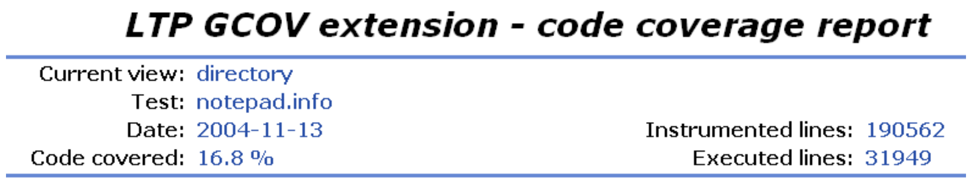
3.2.3 Why We Use LCOV

Essentially, GCOV is unusable with respect to any large project. There are typically too many files and the output is hard to access. In addition, LCOV already has a very reasonable framework in which we can insert our code to do differential code coverage analysis which is one of the major deliverables for this project.

3.2.4 Some Internals of LCOV

LCOV is implemented as five PERL scripts:

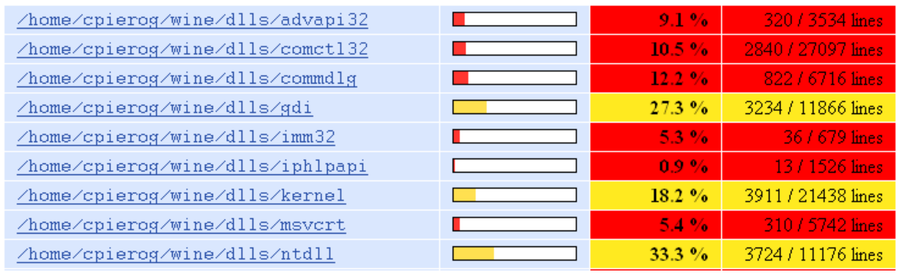
`gendesc` creates test descriptions to be used by genhtml. These descriptions are meant to be used so that one can see which tests are being ran for a given execution. We currently do not use this feature nor did we change this file.



LTP GCOV extension - code coverage report

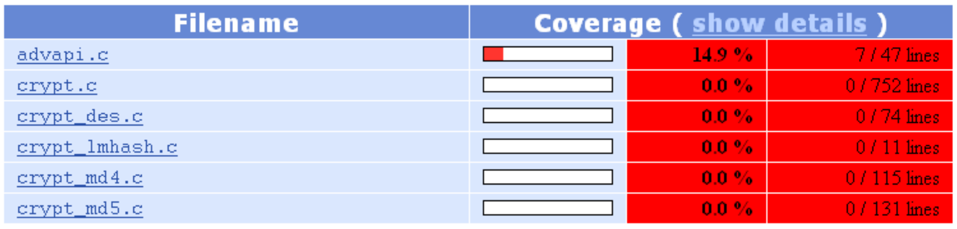
Current view:	directory		
Test:	notepad.info		
Date:	2004-11-13	Instrumented lines:	190562
Code covered:	16.8 %	Executed lines:	31949

Figure 2: Statistics from running Microsoft Notepad on top of Wine



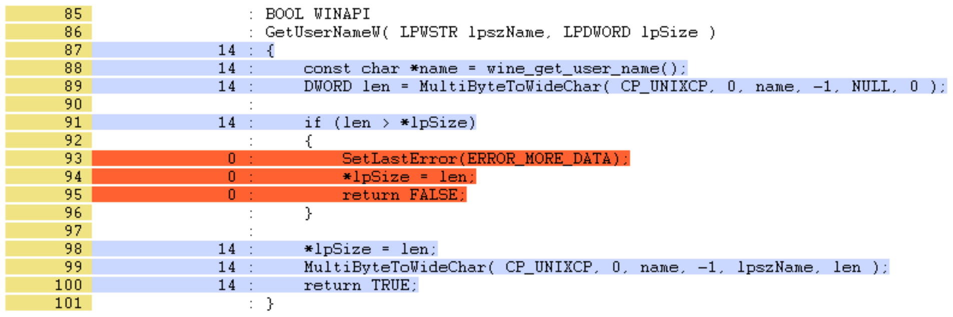
/home/cpierog/wine/dlls/advapi32	9.1 %	320 / 3534 lines
/home/cpierog/wine/dlls/comctl32	10.5 %	2840 / 27097 lines
/home/cpierog/wine/dlls/commdlg	12.2 %	822 / 6716 lines
/home/cpierog/wine/dlls/gdi	27.3 %	3234 / 11866 lines
/home/cpierog/wine/dlls/imm32	5.3 %	36 / 679 lines
/home/cpierog/wine/dlls/iphlpapi	0.9 %	13 / 1526 lines
/home/cpierog/wine/dlls/kernel	18.2 %	3911 / 21438 lines
/home/cpierog/wine/dlls/msvcrt	5.4 %	310 / 5742 lines
/home/cpierog/wine/dlls/ntdll	33.3 %	3724 / 11176 lines

Figure 3: The directories listed in the project



Filename	Coverage (show details)	
advapi.c	14.9 %	7 / 47 lines
crypt.c	0.0 %	0 / 752 lines
crypt_des.c	0.0 %	0 / 74 lines
crypt_lmhash.c	0.0 %	0 / 11 lines
crypt_md4.c	0.0 %	0 / 115 lines
crypt_md5.c	0.0 %	0 / 131 lines

Figure 4: The advapi32 directory's files



```
      : BOOL WINAPI
      : GetUserNameW( LPWSTR lpszName, LPDWORD lpSize )
   14 : {
   14 :     const char *name = wine_get_user_name();
   14 :     DWORD len = MultiByteToWideChar( CP_UNIXCP, 0, name, -1, NULL, 0 );
      :
   14 :     if (len > *lpSize)
      :     {
    0 :         SetLastError(ERROR_MORE_DATA);
    0 :         *lpSize = len;
    0 :         return FALSE;
      :     }
      :
   14 :     *lpSize = len;
   14 :     MultiByteToWideChar( CP_UNIXCP, 0, name, -1, lpszName, len );
   14 :     return TRUE;
      : }
```

Figure 5: A few lines from advapi.c

`genhtml` takes a tracefile ("`.info`" file) and cross references the data to the source file and creates pretty html output. This is the file we modified to accommodate differential code coverage. Actual changes are discussed in Section 3.2.5.

`geninfo` creates the tracefiles from the data obtained from GCOV. This is the part of the project which is the wrapper for recursively applying GCOV.

`genpng` creates a picture of a single source file by using each character as a single picture. We do not currently utilize this tool

`lcov` is wrapper around `geninfo` and also handles many of the options. `lcov` makes the command line interface a little more pleasant by being a front end for the other scripts.

When `lcov` is ran (as in figure 1) it runs `gcov` on every file in every directory recursively. After it has run `gcov` on a file, it then inspects the "`da`" file and creates a set of records for every source line. Once it is done with the current source file, it deletes the "`da`" file and moves to the next file in the directory. Once all files have been looked at, the set of records for each of the source files is written to a tracefile in a format that `genhtml` will understand.

The tracefile format is fairly straight forward. Every line begins with a two letter token which represents what kind of line it is. For our purposes this token can be one of the following:

TN Test Name.

SF Source File. This is specified as an absolute address. So if you attempt rename your directories and then use `genhtml` on a tracefile don't expect any good results.

FN Function. This is a list of functions in the given source file.

DA Data. This is 3-tuple containing information on every line implemented into machine code. This information includes the number of times the line was ran, the line number in the source file, and the a hash of the source line for verification when running `genhtml`.

LF The total number of source lines found in the file.

LH The total number of source lines executed in the file.

Every record is preceded by the `TN` token and ended by the `end_of_record` token. Once the tracefile has been produced by running `lcov`, `genhtml` turns it into a HTML website. This allows for easy navigation and a very nice overview of the results.

3.2.5 Our Changes To LCOV - Differential Code Coverage

Our goal in altering LCOV was to add differential code coverage analysis that would be usable both by the clinic team and the open source community. The philosophy of the team was that we wanted to make the change while altering the source as little as possible.

Since LCOV is implemented in PERL (and none of the team members were perfectly fluent in Perl), it took a little while to acclimate to the new language. Once we became more familiar with the layout of the source files and PERL itself, we added in a new command line option `--nif-baseline` and stuck very close to the standard code of the project. The patch itself can be found in Appendix C.

Our change was to the `genhtml` file. We chose to make all of our changes here since it felt most natural to use two tracefiles (in our case the "`.info`" files) and compare the results of these files. A use case can be found in Section 3.2.6 which may clarify the process.

3.2.6 How To Use Our LCOV with Wine

Our change to LCOV was minimalistic and has no effect if our switch is not used. Only when a developer uses `--nif-baseline` in combination with `-b` (`--baseline`) does one notice the changes we made. As seen below in the use case, the developer creates two tracefiles ("`.info`" files) and then uses `genhtml` to compare the two runs. Note: While you may get some sane results if you use different source trees for the builds, it is strongly encouraged to use the same source tree in order to avoid conflicts when displaying the code in the LCOV html output.

Figure 6: LCOV use case with Wine

```
%> lcov -d testSuite_build -o testSuite.info -c
%> lcov -d GoogleApp_build -o GoogleApp.info -c
%> genhtml -p . -o compare.out -b testSuite.info --nif-baseline -s GoogleApp.info
%> mozilla compare.out/index.html
```

This functionality will be incorporated into the Wine project as soon as the `--enable-gcov` patch is accepted and applied to the current cvs branch. The patch can be found in Appendix B

4 Google Application Testing

Using the code coverage utilities that we have adapted for Wine, we plan to identify portions of the Windows API that the Google applications are using but were not being covered by the Wine conformance test suite. Once we have identified untested areas, we can then fill these gaps in testing, and potentially reveal bugs or underdeveloped areas of code. Thus, we would improve the likelihood of Google applications eventually running seamlessly in Wine.

4.1 Application Testing Process

The following is the process we expect to follow when we begin Google application testing:

1. Compile patched Wine with GCOV code coverage profiling enabled (`./configure --enable-gcov`)
2. Execute Wine conformance test suite using winetest (`./programs/winetest/winetest` or `make -k test`)
3. Execute Google application under Wine (`./wine Picasa.exe`)
4. Utilize patched LCOV to examine code coverage statistics generated by both executions as well as the differences in code coverage between the two executions (see Section 3.2.6)
5. Using LCOV results, identify functionality of the Windows API used by the Google application that remains untested by the Wine conformance test suite
6. Write test files in C or Perl to test newly identified untested Windows API functionality. These tests will first be run under Windows itself to determine the true windows behavior, then they will be executed running under Wine
7. If new tests reveal bugs, submit bug reports to the Wine project
8. Repeat the process

4.2 Target Google Applications

Google has specified four Windows applications for testing in Wine:

- Picasa - a program that indexes, manages and searches through images on a personal computer
- Hello! - a program that facilitates image uploading and posting on blogs
- Google Desktop Search - a program that can searches through common document file formats on a personal computer
- Keyhole - a program which provides geographic satellite maps and geolocation search

Thus far, Picasa has been the main focus of our testing efforts, though we have done cursory probes of the other three applications. Given our initial results with Picasa and the existence of an upcoming beta of the next version, it seems a reasonable candidate for further exploration, and as such, has been identified as our single focus for the near future. In the event that progress slows with Picasa, we can shift our focus to one of the other three applications.

4.3 Initial Results

We have done a cursory examination of the applications running in Wine. The applications we are using for testing are the public versions available for download on the Google website. Currently, we are working with Picasa 1.6, the beta of Google Desktop Search, Keyhole 2 LT and the current version of Hello. Of these applications, Hello is the only application that we have not attempted to execute in any fashion. Any and all anomalies we come across when trying to execute an application in Wine are going to be documented along with any workarounds we discover in the process.

4.3.1 Installation Difficulties

The installers for Picasa, Keyhole and Google Desktop Search each have different failures when being run in Wine. The installer for Picasa repeatedly produces a dialog box with the text "Out of Memory" and an "Ok" button that dismisses the box, only to have it reproduced until the process is ended unnaturally using `kill`. The installer for Keyhole generates a memory access violation and asks if the user would like to debug. The installer for Google Desktop Search produces a dialog box claiming that the user is attempting to install onto an incompatible version of Windows.

In the case of Picasa and Keyhole, it was enough to make a copy of the program's installations and execute the executable with Wine. In the case of Google Desktop Search, a simple copy of directory and executable was not enough. Configuring Wine to act as a different version of Windows did not succeed either. A workaround still hasn't been found yet.

4.3.2 Picasa Slideshow

When running Picasa, attempting to start a slideshow causes the application to hang. LCOV was used to profile Wine while running Picasa and trying to start a slideshow. When the differential code coverage analysis was run on the execution's profiling output and the profiling output from winetest, it was revealed that 601 lines of code were being executed by Picasa and not the test suite. A majority of these lines appear to be in the commctrl.c, datetime.c and monthcal.c files, all part of the commctl32 dll module. To follow through with our process, we plan to write or modify unit tests to expand testing coverage of the functionality in this module.

5 Timeline

5.1 Revised Schedule for Fall '04

	Weeks														
Tasks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Determine Meeting Times	▨														
Preliminary Research	▨	▨	▨	▨											
Clinic Orientation		▨													
Set Up Workstations			▨												
Statement of Work (Outline)			▨												
Statement of Work (Draft)				▨											
Statement of Work (Final)					▨	▨	▨	▨	▨						
First Presentation						▨									
Integration of GCOV into Wine				▨	▨	▨	▨	▨	▨						
LZExpand Tests					▨	▨	▨	▨	▨						
LCOV Diff Code Coverage					▨	▨	▨	▨	▨	▨					
Second Presentation											▨	▨			
Google Application Testing												▨	▨	▨	▨
Mid-Year Report													▨	▨	

5.2 Anticipated Work for Spring '05

5.3 Patches

Our `--enable-gcov` patch for Wine has already been submitted and approved. We would like to accomplish the same for our other two patches.

- `--nif-baseline` patch for LCOV
- `make lcov` patch for Wine

The `make lcov` patch will be submitted to the Wine developers as before. The LCOV patch will be submitted to the Linux Test Project (LTP) who maintain the LCOV code coverage tool.

5.4 Google Applications under Wine

Ideally, we would like to get each of the four Google applications mentioned above to work flawlessly under Wine. However, this is a rather monolithic task. As such, we will start by trying to get most of Picasa's functionality working under Wine. If time permits, we will move onto other applications.

In an attempt to get it working under wine, we anticipate each application to undergo many iterations of the following steps.

- Find an error in the application's performance under Wine. We could automate this process by writing Cxtest scripts to test the GUI for us
- Profile the application when the error is found
- Write unit tests for gaps in test suite that were exposed by the profiling. These unit tests will be submitted to the Wine project.
- Hopefully, these tests will reveal the source of whatever errors we were observing.
- Submit Wine bug reports for newly exposed errors.
- Additionally, we might try to fix a few bugs in Wine ourselves. However, the learning curve on the Wine source might make this infeasible.

A --enable-gcov Patch For Wine

This patch can be found at `http://www.winehq.org/hypermail/wine-patches/2004/11/0136.html` and the documentation can be found at `http://www.winehq.com/site/documentation`.

This patch will likely no longer apply to a current cvs branch; however, it still provides a good idea of what we did and how to do it again should it be necessary.

```
Index: Make.rules.in
===================================================================
RCS file: /home/wine/wine/Make.rules.in,v
retrieving revision 1.179
diff -u -u -r1.179 Make.rules.in
--- Make.rules.in7 Oct 2004 03:12:44 -00001.179
+++ Make.rules.in9 Nov 2004 05:18:32 -0000
@@ -104,8 +104,9 @@
 api_manext      = 3w
 conf_manext     = 5
 CLEAN_FILES     = *.o *.a *.so *.ln *.$(LIBEXT) \\\#*\\\# *~ *% .\\\#* *.bak *.orig *.r
-                  *.flc *.spec.c *.spec.def *.dbg.c *.tab.c *.tab.h @LEX_OUTPUT_ROOT@.c
+                  *.flc *.spec.c *.spec.def *.dbg.c *.tab.c *.tab.h @LEX_OUTPUT_ROOT@.c
+                  *.bb *.bbg *.da

 OBJS = $(C_SRCS:.c=.o) $(EXTRA_OBJS)


Index: configure.ac
===================================================================
RCS file: /home/wine/wine/configure.ac,v
retrieving revision 1.324
diff -u -u -r1.324 configure.ac
--- configure.ac4 Nov 2004 21:15:33 -00001.324
+++ configure.ac9 Nov 2004 05:18:32 -0000
@@ -16,6 +16,7 @@
 AC_ARG_ENABLE(debug, AC_HELP_STRING([--disable-debug],[compile out all debugging messag
 AC_ARG_ENABLE(trace, AC_HELP_STRING([--disable-trace],[compile out TRACE messages]))
 AC_ARG_ENABLE(win64, AC_HELP_STRING([--enable-win64], [build a Win64 emulator on AMD64
+AC_ARG_ENABLE(gcov, AC_HELP_STRING([--enable-gcov],[turn on code coverage analysis tool

 AC_ARG_WITH(opengl,    AC_HELP_STRING([--without-opengl],[do not use OpenGL]))
 AC_ARG_WITH(curses,    AC_HELP_STRING([--without-curses],[do not use curses]))
@@ -707,6 +708,7 @@
 dnl **** Check for gcc specific options ****
 AC_SUBST(EXTRACFLAGS,'''')
+AC_SUBST(LOADEREXTRACFLAGS,'''')
 if test ``x${GCC}'' = ``xyes''
 then
   EXTRACFLAGS=''-Wall -pipe''
@@ -768,6 +770,31 @@
     EXTRACFLAGS=''$EXTRACFLAGS -gstabs+''
   fi

+  dnl Check for --enable-gcov and add appropriate flags for gcc
+  dnl Note that these extra switches are NOT applied to the loader
+  if test ``x$enable_gcov'' = ``xyes'';
+  then
+    dnl Check for -fprofile-arcs and -ftest-coverage option
+    AC_CACHE_CHECK([for gcc -fprofile-arcs support],
+    ac_cv_c_gcc_fprofile_arcs,
+                    [WINE_TRY_CFLAGS([-fprofile-arcs],
+                     ac_cv_c_gcc_fprofile_arcs=''yes'',
+     ac_cv_c_gcc_fprofile_arcs=''no'')])
+    AC_CACHE_CHECK([for gcc -ftest-coverage support],
+    ac_cv_c_gcc_ftest_coverage,
+                    [WINE_TRY_CFLAGS([-ftest-coverage],
+                     ac_cv_c_gcc_ftest_coverage=''yes'',
+     ac_cv_c_gcc_ftest_coverage=''no'')])
+    if test ``$ac_cv_c_gcc_fprofile_arcs'' = ``yes'' && \
+       test ``$ac_cv_c_gcc_ftest_coverage'' = ``yes''
+    then
+      EXTRACFLAGS=''$EXTRACFLAGS -fprofile-arcs -ftest-coverage''
+      dnl Turn off optimization so code coverage tool
+      dnl can get accurate line numbers
+      EXTRACFLAGS=`echo ``$EXTRACFLAGS'' | sed -e 's/-O[0-9]*//g'`
+    fi
+  fi
+
   dnl Check for noisy string.h
   saved_CFLAGS=''$CFLAGS''
   CFLAGS=''$CFLAGS -Wpointer-arith -Werror''
@@ -781,6 +808,11 @@
   fi
 fi
```

```
+dnl When adding in more 'EXTRAFLAGS' you must put them above this!
+dnl Strips out the code profiling abilities because the preloader
+dnl relies on low level assumptions which added code messes up.
+LOADEREXTRACFLAGS=`echo ``$EXTRACFLAGS'' | sed -e 's/-fprofile-arcs -ftest-coverage//g'
+
 dnl **** Check how to define a function in assembly code ****

 AC_CACHE_CHECK([how to define a function in assembly code], ac_cv_asm_func_def,
Index: documentation/winedev-otherdebug.sgml
===================================================================
RCS file: /home/wine/wine/documentation/winedev-otherdebug.sgml,v
retrieving revision 1.1
diff -u -u -r1.1 winedev-otherdebug.sgml
--- documentation/winedev-otherdebug.sgml26 Oct 2004 22:45:47 -00001.1
+++ documentation/winedev-otherdebug.sgml9 Nov 2004 05:18:32 -0000
@@ -497,6 +497,179 @@
 </listitem>
       </orderedlist>
     </sect1>
+    <sect1>
+      <title>Which code has been tested?</title>
+      <para>
+        Deciding what code should be tested next can be a difficult
+        decision.  And in any given project, there is always code that
+        isn't tested where bugs could be lurking.  This section goes
+        over how to identify these sections using a tool called gcov.
+      </para>
+      <para>
+        To use gcov on wine, do the following:
+      </para>
+      <orderedlist>
+        <listitem>
+         <para>
+           In order to activate code coverage in the wine source
+           code, use the <literal>--enable-gcov</literal> flag when
+           running <command>configure</command>.
+         </para>
+       </listitem>
+        <listitem>
+         <para>
+           Run any application or test suite.
+         </para>
+       </listitem>
+        <listitem>
+         <para>
+           Run gcov on the file which you would like to know more
+           about code coverage.
+         </para>
+       </listitem>
+      </orderedlist>
+      <para>
+        The following is an example situation when using gcov to
+        determine the coverage of a file could be helpful.  We'll use
+        the <filename>dlls/lzexpand/lzexpand_main.c.</filename> file.
+        At one time the code in this file was not fully tested (as it
+        may still be).  For example at the time of this writing, the
+        function <function>LZOpenFileA</function> had the following
+        lines in it:
+        <screen>
+if ((mode&~0x70)!=OF_READ)
+        return fd;
+if (fd==HFILE_ERROR)
+        return HFILE_ERROR;
+cfd=LZInit(fd);
+if ((INT)cfd <= 0) return fd;
+return cfd;
+        </screen>
+        Currently there are a few tests written to test this function;
+        however, these tests don't check that everything is correct.
+        For instance, <constant>HFILE_ERROR</constant> may be the wrong
+        error code to return.  Using gcov and directed tests, we can
+        validate the correctness of this line of code.  First, we see
+        what has been tested already by running gcov on the file.
+        To do this, do the following:
+        <screen>
+cvs checkout wine
+mkdir build
+cd build
+../wine/configure --enable-gcov
+make depend && make
+cd dlls/lxexpand/tests
+make test
+cd ..
+gcov ../../../wine/dlls/lzexpand/lzexpand_main.c
```

```
+  0.00% of 3 lines executed in file ../../../wine/include/wine/unicode.h
+  Creating unicode.h.gcov.
+  0.00% of 4 lines executed in file /usr/include/ctype.h
+  Creating ctype.h.gcov.
+  0.00% of 6 lines executed in file /usr/include/bits/string2.h
+  Creating string2.h.gcov.
+  100.00% of 3 lines executed in file ../../../wine/include/winbase.h
+  Creating winbase.h.gcov.
+  50.83% of 240 lines executed in file ../../../wine/dlls/lzexpand/lzexpand_main.c
+  Creating lzexpand_main.c.gcov.
+less lzexpand_main.c.gcov
+        </screen>
+        Note that there is more output, but only output of gcov is
+        shown.  The output file
+        <filename>lzexpand_main.c.gcov</filename> looks like this.
+        <screen>
+        9:  545:        if ((mode&~0x70)!=OF_READ)
+        6:  546:                return fd;
+        3:  547:        if (fd==HFILE_ERROR)
+    #####:  548:                return HFILE_ERROR;
+        3:  549:        cfd=LZInit(fd);
+        3:  550:        if ((INT)cfd <= 0) return fd;
+        3:  551:        return cfd;
+        </screen>
+        <command>gcov</command> output consists of three components:
+        the number of times a line was run, the line number, and the
+        actual text of the line.  Note: If a line is optimized out by
+        the compiler, it will appear as if it was never run.  The line
+        of code which returns <constant>HFILE_ERROR</constant> is
+        never executed (and it is highly unlikely that it is optimized
+        out), so we don't know if it is correct.  In order to validate
+        this line, there are two parts of this process.  First we must
+        write the test.  Please see <xref linkend=''testing''> to
+        learn more about writing tests.  We insert the following lines
+        into a test case:
+        <screen>
+INT file;
+
+/* Check for non-existent file. */
+file = LZOpenFile(``badfilename_'', &amp;test, OF_READ);
+ok(file == LZERROR_BADINHANDLE,
+   ``LZOpenFile succeeded on nonexistent file\n'');
```

```
+LZClose(file);
+       </screen>
+        Once we add in this test case, we now want to know if the line
+        in question is run by this test and works as expected.  You
+        should be in the same directory as you left off in the previous
+        command example.  The only difference is that we have to remove
+        the <filename>*.da</filename> files in order to start the
+        count over (if we leave the files than the number of times the
+        line is run is just added, e.g. line 545 below would be run 19 times)
+        and we remove the <filename>*.gcov</filename> files because
+        they are out of date and need to be recreated.
+       </para>
+       <screen>
+rm *.da *.gcov
+cd tests
+make
+make test
+cd ..
+gcov ../../../wine/dlls/lzexpand/lzexpand_main.c
+  0.00% of 3 lines executed in file ../../../wine/include/wine/unicode.h
+  Creating unicode.h.gcov.
+  0.00% of 4 lines executed in file /usr/include/ctype.h
+  Creating ctype.h.gcov.
+  0.00% of 6 lines executed in file /usr/include/bits/string2.h
+  Creating string2.h.gcov.
+  100.00% of 3 lines executed in file ../../../wine/include/winbase.h
+  Creating winbase.h.gcov.
+  51.67% of 240 lines executed in file ../../../wine/dlls/lzexpand/lzexpand_main.c
+  Creating lzexpand_main.c.gcov.
+less lzexpand_main.c.gcov
+       </screen>
+       <para>
+         Note that there is more output, but only output of gcov is
+         shown.  The output file
+         <filename>lzexpand_main.c.gcov</filename> looks like this.
+       </para>
+       <screen>
+        10:  545:        if ((mode&~0x70)!=OF_READ)
+         6:  546:                return fd;
+         4:  547:        if (fd==HFILE_ERROR)
+         1:  548:                return HFILE_ERROR;
+         3:  549:        cfd=LZInit(fd);
+        3:  550:        if ((INT)cfd <= 0) return fd;
+        3:  551:        return cfd;
+      </screen>
+      <para>
+        Based on gcov, we now know that
+        <constant>HFILE_ERROR</constant> is returned once.  And since
+        all of our other tests have remain unchanged, we can assume
+        that the one time it is returned is to satisfy the one case we
+        added where we check for it.  Thus we have validated a line of
+        code.  While this is a cursory example, it demostrates the
+        potential usefulness of this tool.
+      </para>
+      <para>
+        For a further in depth description of gcov, the official gcc
+        compiler suite page for gcov is <ulink
+        url=''http://gcc.gnu.org/onlinedocs/gcc-3.2.3/gcc/Gcov.html''>
+        http://gcc.gnu.org/onlinedocs/gcc-3.2.3/gcc/Gcov.html</ulink>.
+        There is also an excellent article written by Steve Best for
+        Linux Magazine which describes and illustrates this process
+        very well at
+        <ulink url=''http://www.linux-mag.com/2003-07/compile_01.html''>
+        http://www.linux-mag.com/2003-07/compile_01.html</ulink>.
+      </para>
+    </sect1>

   </chapter>
Index: loader/Makefile.in
===================================================================
RCS file: /home/wine/wine/loader/Makefile.in,v
retrieving revision 1.17
diff -u -u -r1.17 Makefile.in
--- loader/Makefile.in11 Aug 2004 20:59:09 -00001.17
+++ loader/Makefile.in9 Nov 2004 05:18:32 -0000
@@ -23,6 +23,10 @@

 LIBPTHREAD  = @LIBPTHREAD@
 LDEXECFLAGS = @LDEXECFLAGS@
+LOADEREXTRACFLAGS = @LOADEREXTRACFLAGS@
+
+.c.o:
+$(CC) -c $(INCLUDES) $(DEFS) $(DLLFLAGS) $(LOADEREXTRACFLAGS) $(CPPFLAGS) $(CFLAGS) -o
```

```
wine-glibc: glibc.o Makefile.in
$(CC) -o $@ glibc.o $(LIBWINE) $(LIBPORT) $(LIBPTHREAD) $(EXTRALIBS) $(LDFLAGS)
```

B make lcov Patch For Wine

```
Index: Make.rules.in
===================================================================
RCS file: /home/wine/wine/Make.rules.in,v
retrieving revision 1.179
diff -u -u -r1.179 Make.rules.in
--- Make.rules.in7 Oct 2004 03:12:44 -00001.179
+++ Make.rules.in4 Dec 2004 06:35:23 -0000
@@ -81,6 +81,9 @@
 LIBPORT      = -L$(TOPOBJDIR)/libs/port -lwine_port
 LIBUNICODE   = -L$(TOPOBJDIR)/libs/unicode -lwine_unicode
 LIBWINE      = -L$(TOPOBJDIR)/libs/wine -lwine
+LCOV_OUTPUT  = $(TOPOBJDIR)/lcov.out
+LCOV_LCOV    = @LCOV@
+LCOV_GENHTML = @GENHTML@

 @SET_MAKE@

Index: Makefile.in
===================================================================
RCS file: /home/wine/wine/Makefile.in,v
retrieving revision 1.159
diff -u -u -r1.159 Makefile.in
--- Makefile.in7 Oct 2004 03:12:44 -00001.159
+++ Makefile.in4 Dec 2004 06:35:23 -0000
@@ -14,6 +14,8 @@
 # manpages:        compile manpages for Wine API
 # htmlpages:       compile html pages for Wine API
 # sgmlpages:       compile sgml source for the Wine API Guide
+# lcov:            run lcov to gain code coverage abilities
+# lcov-clean:       remove directory with lcov data

 # Directories

@@ -152,6 +154,14 @@
 sgmlpages:
 $(MKINSTALLDIRS) $(TOPOBJDIR)/documentation/api-guide
 cd dlls && $(MAKE) doc-sgml
+lcov:
+$(LCOV_LCOV) --directory $(TOPOBJDIR) --capture --output-file wine.info --test-name WIN
+LANG=C $(LCOV_GENHTML) --prefix . --output-directory $(LCOV_OUTPUT) --title ‘‘WINE Code
```

```
+@echo ‘‘Point a web browser at $(LCOV_OUTPUT)/index.html to see results.’’
+
+lcov-clean:
+$(RM) -r $(LCOV_OUTPUT)
+

 clean::
 $(RM) wine
```

C --nif-baseline Patch For LCOV

```
Index: utils/analysis/lcov/bin/genhtml
===================================================================
RCS file: /cvsroot/ltp/utils/analysis/lcov/bin/genhtml,v
retrieving revision 1.7
diff -u -r1.7 genhtml
--- utils/analysis/lcov/bin/genhtml9 Aug 2004 11:15:02 -00001.7
+++ utils/analysis/lcov/bin/genhtml30 Oct 2004 04:42:56 -0000
@@ -177,6 +177,7 @@
 our $keep_descriptions;# If set, do not remove unused test case descriptions
 our $no_sourceview;# If set, do not create a source code view for each file
 our $highlight;# If set, highlight lines covered by converted data only
+our $nif_baseline;       # Do we do boolean baseline compare code coverage?
 our $tab_size = 8;# Number of spaces to use in place of tab
 our $config;# Configuration file contents

@@ -229,6 +230,7 @@
 ‘‘keep-descriptions’’=> \$keep_descriptions,
 ‘‘css-file=s’’=> \$css_filename,
 ‘‘baseline-file=s’’=> \$base_filename,
+’’nif-baseline’’  => \$nif_baseline,
 ‘‘prefix=s’’=> \$dir_prefix,
 ‘‘num-spaces=i’’=> \$tab_size,
 ‘‘no-prefix’’=> \$no_prefix,
@@ -359,6 +361,7 @@
   -s, --show-details                Generate detailed directory view
   -f, --frames                      Use HTML frames for source code view
   -b, --baseline-file BASEFILE      Use BASEFILE as baseline file
+  -n, --nif-baseline                Do boolean baseline (normally subtraction)
   -o, --output-directory OUTDIR     Write HTML output to OUTDIR
   -t, --title TITLE                 Display TITLE in header of all pages
   -d, --description-file DESCFILE   Read test case descriptions from DESCFILE
@@ -2900,7 +2903,7 @@
 }

 open(SOURCE_HANDLE, ‘‘<’’.$source_filename)
-or die(‘‘ERROR: cannot open $source_filename for reading!\n’’);
+or print(‘‘ERROR: cannot open $source_filename for reading!\n’’);

 write_source_prolog(*HTML_HANDLE);
```

```
@@ -2912,7 +2915,7 @@
 if (defined($checkdata->{$line_number}) &&
     ($checkdata->{$line_number} ne md5_base64($_)))
 {
-die(‘‘ERROR: checksum mismatch  at $source_filename:’’.
+print(‘‘ERROR: checksum mismatch  at $source_filename:’’.
     ‘‘$line_number\n’’);
 }

@@ -2948,6 +2951,9 @@
 #
 # subtract_counts(data_ref, base_ref)
 #
+# Subtract base hash from data hash.
+# If --nif-baseline is used, then give a count of 1 if the data hash
+# contains the line and the base hash did not, and 0 otherwise.

 sub subtract_counts($$)
 {
@@ -2969,14 +2975,25 @@
 {
 $data_count -= $base_count;

-# Make sure we don’t get negative numbers
-if ($data_count<0) { $data_count = 0; }
+if ($nif_baseline)
+{
+# Set data count (see above)
+$data_count = ($base_count == 0 &&
+       $data_count > 0) ? 1 : 0;
+}
 }
-
-$data{$line} = $data_count;
+elsif ($nif_baseline)
+{
+# Line is in data hash and not the base hash
+$data_count = 1;
+}
+
+# Make sure we don’t get negative numbers
+if ($data_count < 0) { $data_count = 0; }
```

```
 if ($data_count > 0) { $hit++; }
+$data{$line} = $data_count;
 }
-
+
 return (\%data, $found, $hit);
 }

@@ -3074,6 +3091,14 @@
 ($base_testdata, $base_count, undef, $base_checkdata) =
 get_info_entry($base);

+# if there is no file!!!
+if (!defined($base_count))
+{
+print(``NO BASE COUNT FOR $filename!!!'');
+next;
+}
+
+
 # Check for compatible checksums
 merge_checksums($data_checkdata, $base_checkdata, $filename);

@@ -3086,6 +3111,15 @@
 # Get counts of both data and baseline
 $data_count = $data_testdata->{$testname};

+
+# if there is no file!!!
+if (!defined($data_count))
+{
+print(``NO DATA COUNT FOR $filename!!!'');
+next;
+}
+
+
 $hit = 0;

 ($data_count, undef, $hit) =
Index: utils/analysis/lcov/man/genhtml.1
===================================================================
RCS file: /cvsroot/ltp/utils/analysis/lcov/man/genhtml.1,v
```

```
retrieving revision 1.3
diff -u -r1.3 genhtml.1
--- utils/analysis/lcov/man/genhtml.19 Aug 2004 11:15:02 -00001.3
+++ utils/analysis/lcov/man/genhtml.130 Oct 2004 04:42:56 -0000
@@ -14,6 +14,8 @@
 .RB [ \-b | \-\-baseline\-file  ]
 .IR baseline\-file
 .br
+.RB [ \-\-nif\-baseline ]
+.br
 .RB [ \-o | \-\-output\-directory
 .IR output-directory ]
 .br
@@ -139,6 +141,29 @@
 the result is zero.

 .RE
+.BI ``\-\-nif\-baseline ``
+.RS
+Use boolean compare when applying the baseline.
+
+When the baseline is applied, this flag changes the counts on each
+line in the following manner. If the line was run in the original
+.IR tracefile ,
+but not in the
+.I baseline\-file
+then the line will output with an associated count of 1. Otherwise,
+the line count will be 0.
+
+This option is useful for quickly determining what lines are run in
+the original
+.I tracefile
+that are not run in the
+.IR baseline\-file .
+
+This option is ignored when not used in conjuction with
+.BR \-\-baseline-file .
+
+
+.RE
 .BI ``\-o `` output\-directory
 .br
```

```
.BI ‘‘\-\-output\-directory ‘‘ output\-directory
```