So Optimization Breaks Your Code

R.K. Owen, National Energy Research Scientific Computing Center, NERSC / MS 50C, One Cyclotron Road, Berkeley,CA 94720

ABSTRACT: *Many compiler options can have an adverse affect on a code. Mostnotably, various levels of optimization can "break" a code, such that the program gives erroneous results. One option is to compile for the lowest common denominator. The other option is to identify which routines are adversely affected. The local utility, bchop performs a binary chop between object files in two directories (./good and ./bad), compares the output and isolates which object files are causing the differences in output. This helps isolate problems due to changing compiler or preprocessor options. The number of runs performed is approximately 2*E*log2(N) where N is the number of object files and E is the number of object modules causing errors. This utility, bchop, is similar to CRI's atchop for isolating multitasking problems.*

Introduction

Usually the initial phases of code development rely heavily on a debug-compile-fix scenario. It is not until after the code has been debugged that thoughts of using compiler optimization options surface. Far too often, optimization "breaks" the code and the discovery process has to begin again. Re-compiling to allow easy symbolic debugging requires optimization to be turned off. Therefore, the hapless code developer is at a disadvantage in finding which parts of the program are adversely affected by optimization.

In another scenario, the code is debugged, optimized, and verified. Two or three years pass by, the system compiler is upgraded several times as well as the operating system; or the code is taken to a similar, but not quite identical system. Recompiling with full optimization now results in erroneous results that are hard to trace down. One option is to recompile with optimization turned off, but at a substantial performance penalty.

The ideal solution is to find which compilation units, as well as the trouble spots within them, are adversely affected by compiler optimization levels. In the absence of an ideal solution bchop can at least find which compilation units are thus affected. This method then favors modular programming such that each and every subroutine or function is isolated into it's own compilation unit.

Theory

What little theory there is in bchop is summarized by the phrase "binary decomposition"[1] or sometimes refered to as "interval halving". The procedure is to group half of the compilation units (or object files) into one set and the remaining objects into another set. The first set will comprise object files collected from the object files giving "bad" results and the remainder will be collected from the object files giving "good" results. In practice, the "bad" object files are those compiled with optimization turned on and the "good" object files are compiled with debugging options which usually inhibit optimization. The object files are compiled once only by the user and placed into directories named ./bad and ./good respectively. The "good" and "bad" object files are linked together, the binary is then executed and the output is compared to an accepted "good" output. If the output compares favorably then those object files in the "bad" set are tagged as acceptable and are no longer considered for further testing. On the other hand, if the output shows a difference the set of "bad" object files is further divided in half and the above procedure is repeated. Each object file represents an endpoint in a binary tree, which takes O(logN) steps to traverse.

Practice

The above algorithm describes a depth first search. For practical reasons bchop does a lateral search proceeding from the root downwards. The object files are numbered and selected into sets according to the least significant bits. The first level corresponds to an odd/even split. The odd numbered set of object files are tested, then the even numbered set is tested. If one set is deemed acceptable the rest of the object files are "reshuffled" (i.e. reordered and renumbered), and the procedure is repeated with the reduced set. However, if both sets yield incorrect results the collection of object files is split into fourths, and each fourth is tested. If there is still no resolution the collection of object files is split into eighths, and so forth. See Figure 1 for a simple example of how the sets are selected.

When any of the sets contain only one element then bchop goes into a one-by-one comparison mode where each object file is tested individually. It will be shown later that one-by-one testing is more efficient for small samples or when the ratio of "bad" to "good" object files is large.

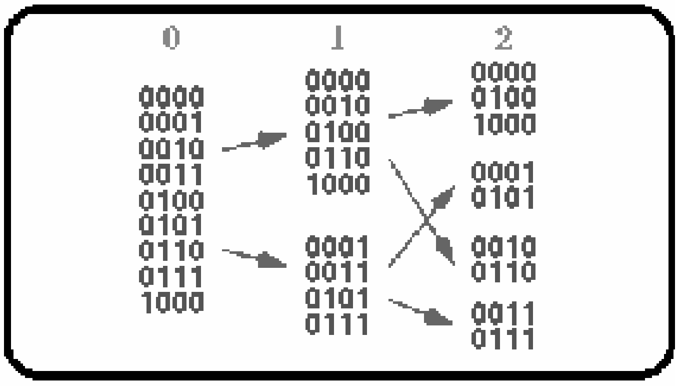

Figure 1: A binary decomposition of 9 objects selected by the least significant binary digits.

Number of Trials

The bchop utility can only be useful if the number of trials or test runs is found to be smaller than the number of object files. Otherwise, it would be more efficient to perform a one-by-one testing, that is - substitute a single "bad" object file in, reload, execute, compare output, and repeat for another object file. Each object in the binary decomposition is a branch in a binary tree. The number of trials is proportional to O(logN) steps to perform. If it was known apriori that there was only one misbehaving object file the number of trials would be less than or equal to log2(N). Unfortunately, the number of errant object files is usually not known apriori and can only be deduced after the entire ensemble is tested. Therefore, bchop can not assume knowledge that could speed up the discovery process. Figure 2 shows bchop testing each set and fine tuning the search. The top bar in the figure indicates that bchop always performs a trial with all the "good" object files as well as all the "bad" object files. This verifies that bchop can discern the difference[1]. A collection of object files are no longer tested if they produce correct results. The number of trials will be exactly equal to 2log2(N), when N is a power of 2. Generally, the number of trials is less and depends what order the "bad" object files are in and when the one-by-one comparison provision is invoked for sets that become "too" finely divided.

When more than one errant object file exists, say E of them, the number of trials can be shown to be less than or equal to

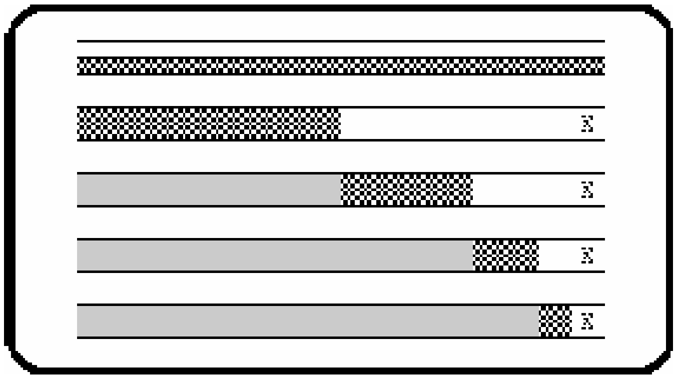

Figure 2: A bchop binary decomposition where X represents a `bad" object file. Red represents tests with errant results, and pink for tests with correct results. Further testing is no longer performed on object files that give correct results.

2Elog2(N). Refer to Figure 3 for an example of two "bad" object files out of 16. Since both the errant object files lie in the same half of the ordering at the second level of testing both tests indicate incorrect results, thus requiring the sample to be more finely divided and retested.

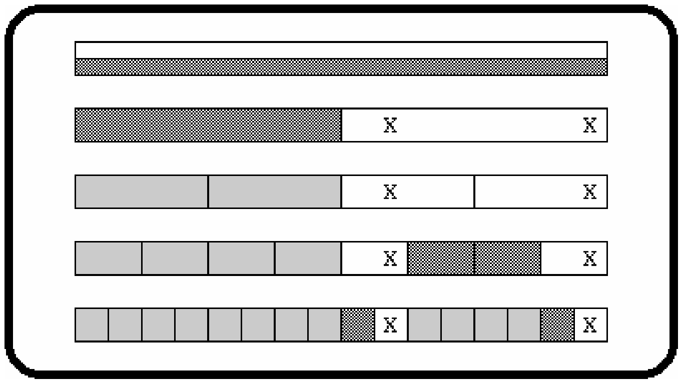

Figure 3: A bchop binary decomposition where the X's represents "bad" object files, see Figure 2 for details regarding the color scheme. The two "bad" object files lie in the same half of the ordering.

A more extensive test was performed with up to 140 object files and up to three errors randomly scattered among them. The test was repeated three times for each configuration. The results are plotted in Figure 4. There is definitely good agreement with the number of trials performed and the theoretical upper limit. However, for small numbers of object files (generally less than 15) it is usually more efficient to perform a one-by-one comparison test unless it is known before hand that the number of errant object files is very small. The actual number of trials performed is highly dependent on errant object file location in the ordered list. If, for example, they all fell into the same sampling octant then the number of trials performed would be comparable to a task an eighth of the size. The larger the number of "bad" object files, the more likely they will demonstrate clustering.

[1] footnote_one_herebchop is quite flexible in allowing the user to include a custom script for output comparison. The output can be filtered to extract only the relevant data, skipping date and time fields which by their very nature will vary from run to run.

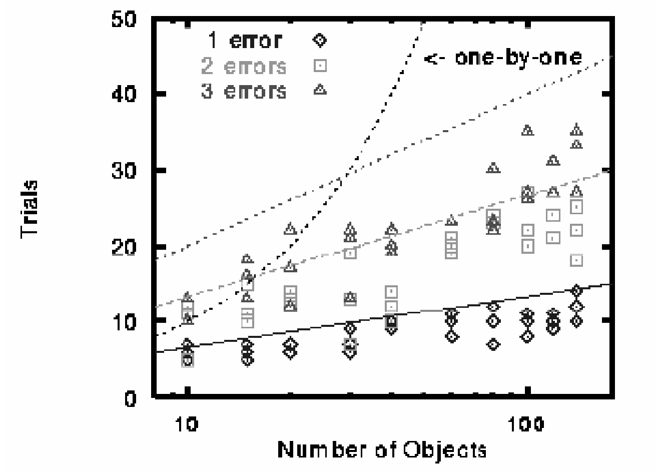

Figure 1: bchop results for 1,2, and 3 “bad” object files in a large number of object files. The curved line represents the number of trials needed to perform a one-by-one comparison test. The upper limit for the number of trials is given by 2Elog2(N), where N is the total number of object files and E is the number of “bad” object files. The straight lines represent this upper limit given for E=3,2, and 1 from the top down respectively.

Implementation Details

The code for bchop is written in “Standard C”[2] and has compiled without incident on Crays, workstations (using the GNU gcc compiler), and PC's running Linux. The program primarily invokes the Standard C system() function call to perform the various tasks of calling the loader, executing the trial binary, and comparing output results. A portion of the program performs the bookkeeping necessary for identifying which object files were tested or not, and whether they passed or failed. The bulk of the program, however, handles the tedious task of pasting character strings together to create command strings to be passed to the system() call. The bchop utility allows for almost every aspect of the load, execution input/output, and comparison stages to be customized with command line options; but assumes certain defaults indicative of a C code in a Unix environment. The source code for bchop is freely available from ftp://ftp.kudonet.com/pub/vip/rkowen/ and follows the same general copyright as the GNU software. The author is continuing to improve the bchop utility and invites any suggestions (and welcomes any code improvements) to be sent to rk@owen.sj.ca.us.

Conclusion

The bchop utility is a simple, but effective tool for isolating which compilation unit may be adversely affected by changes in compiler options, system environment, compiler changes, etc. The original task that prompted the creation of this utility was a Fortran program with 117 compilation units that had successfully compiled and ran at another supercomputer center earlier in the program's development cycle, but was now giving erroneous results. Cray's atchop[3,4] utility was investigated as a tool to diagnose the problem, but it quickly became apparent that 1) it required recompilation of the entire code for each test, 2) was specific to isolating multitasking problems only, and 3) proved difficult to use. It was found that two subroutines after 23 trials (and approximately two Cray C90 CPU hours) were adversely affected by optimization. However, one of the routines identified gave incorrect results in a part of the output that was not considered or even looked at by the user. Approximately three days was spent in developing and debugging this utility (amongst the author's various other duties) to a point where it identified the user's problem routines. It has been successfully used several times since then by the author and others to track down problems resulting from compiler upgrades.

Bibliography

1. G.H. Gonnet, R. Baeza-Yates, “Handbook of Algorithms and Data Structures: in Pascal and C, 2nd Edition”, Addison-Wesley Publishing Co., 1991.
2. P.J. Plauger, Jim Brodie, “Standard C: Programmer's Quick Reference”, Microsoft Press, 1989.
3. UNICOS Performance Utilities Reference Manual, SR-2040 8.0, Cray Research, Inc.
4. UNICOS 8.0 atchop man page, Cray Research, Inc.