Effectively Prioritizing Tests in Development Environment

Amitabh Srivastava
Microsoft Research
One Microsoft Way
Redmond, WA
amitabhs@microsoft.com

Jay Thiagarajan
Microsoft Research
One Microsoft Way
Redmond, WA
jaythia@microsoft.com

ABSTRACT

Software testing helps ensure not only that the software under development has been implemented correctly, but also that further development does not break it. If developers introduce new defects into the software, these should be detected as early and inexpensively as possible in the development cycle. To help optimize which tests are run at what points in the design cycle, we have built *Echelon*, a test prioritization system, which prioritizes the application's given set of tests, based on what changes have been made to the program.

Echelon builds on the previous work on test prioritization and proposes a practical binary code based approach that scales well to large systems. Echelon utilizes a binary matching system that can accurately compute the differences at a basic block granularity between two versions of the program in binary form. Echelon utilizes a fast, simple and intuitive heuristic that works well in practice to compute what tests will cover the affected basic blocks in the program. Echelon orders the given tests to maximally cover the affected program so that defects are likely to be found quickly and inexpensively. Although the primary focus in Echelon is on program changes, other criteria can be added in computing the priorities.

Echelon is part of a test effectiveness infrastructure that runs under the Windows environment. It is currently being integrated into the Microsoft software development process. Echelon has been tested on large Microsoft product binaries. The results show that Echelon is effective in ordering tests based on changes between two program versions.

Keywords

Software testing, test prioritization, regression testing, test selection, test minimization.

1. INTRODUCTION

In large-scale software development, testing accounts for a substantial portion of the development cost. An important goal of testing is to expose defects in software; detection early in the development cycle saves time and resources. Testing of software, therefore, occurs continuously throughout the development cycle[1]. For example, developers may run a few simple pre-checkin tests to ensure that their code changes will not keep the program from being built (compiled and linked) and to catch a moderate number of defects. Later, after the whole program is built, verification tests are run before it is released for full testing. These tests are not intended to be exhaustive and must complete within a limited time.

Full testing, running all tests in the test suite, is intended to be exhaustive and may take days or weeks to run. Even during full testing, it remains advantageous to detect the defects as early as possible, e.g., on day 1 rather than day 21. Early detection of defects enables developers to start sooner on finding and fixing defects for the next iteration. Once the software is released, software patches to update released software also go through the regular test process. However, there are certain rare circumstances, such as emergency patches for critical bugs, when tests must be run under severe time constraints and certain tests must be skipped.

To address these scenarios effectively, developers and testers must be able to run the right tests at the right time. New defects recently introduced into the system are most likely to be from recent code changes. Therefore, an effective strategy is to focus testing efforts on parts of the program affected by changes. Whenever a developer checks in code, a set of tests can be dynamically selected to exercise parts of the program affected by the developer's code changes, subject to a specified time limit. This same technique can also be applied later to testing the system after it is built. Even for full testing, we first want to run tests that will exercise the affected parts of the program, before any other tests.

For any of these techniques to be used in a large-scale development environment, of course, they must be fast, useful, and integrated into the development process.

Over the past decade a number of techniques have been proposed and we discuss them in Section 2. We have built a test prioritization system, Echelon, which prioritizes an existing set of tests to address all these scenarios. Echelon extends the previous techniques, and proposes a practical binary code based approach

[1] For this discussion, we'll use the following simple development model. Developers write code based on the program's specification and check it into the development process. Code is compiled and linked and the application is built, verified, and then released for testing. Defects (program not behaving as per its specification) detected during testing are fixed in the code, the fixes are checked in, and this process is repeated until all detected defects have been eliminated.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

for test prioritization based on program change. It takes as input two versions of the program in binary form along with the test coverage information of the older version, detailing which tests cover which parts of the program. Echelon outputs a prioritized list of tests. This ordered list of tests starts with a minimal sequence of tests, drawn from the given tests, that cover as much of the affected program as possible[2]. This is followed by another minimal sequence of tests drawn from the remaining tests, and so on; it ends with a sequence of tests that do not cover any of the affected parts of the program. Even though all the subsequent sequences do not cover any new block that has not already been covered by the first sequence, it is still beneficial to run as many sequences as possible because they might cover the same code with different data values.

It is important to emphasize that Echelon does not eliminate any tests itself; it merely prioritizes them, so as much of the changed code is tested as early as possible. In time-constrained environments, of course, an appropriate number of test sequences can be selected from the prioritized list, and Echelon provides a variety of information to guide this decision.

The key features of Echelon are:

- Echelon prioritizes tests into an ordered sequence of tests based on program change.
- Echelon compute changes between programs at a very fine granularity—at the level of basic blocks— using an accurate binary matching algorithm.
- Echelon uses a fast, simple algorithm that works well in practice and does not attempt an expensive data flow algorithm to determine which new code will be covered by which existing tests.
- Echelon operates at the binary level making it easier to integrate into the development process. Echelon scales to real product binaries in large-scale development environments.
- Echelon also produces a list of program source code that will not be covered by any of the given existing tests. New tests will be needed to test them.

This paper discusses Echelon. It describes the previous work and how it relates to Echelon, the test prioritization system, and Echelon's implementation. The paper presents the results from prioritizing tests on binaries from large-scale development environments, and discusses Echelon's effectiveness.

2. RELATED WORK

To address the cost of testing, the following approaches have been proposed: test selection [1][3][5][20][28], test prioritization [6][7][8][17][18], and a hybrid approach [31] combining the two previous techniques by starting first with test selection using source code changes [14] followed by test prioritization to schedule the selected tests. Test minimization techniques that permanently discard tests have been proposed in [4][13][19][30].

[2] Some changes especially new code may not be covered by any existing tests. Echelon enumerates these exceptions.

As Echelon applies prioritization in context of knowledge of the program change, we will focus our discussion on techniques that use program change. Program change has been proposed [1][3][4][20][28] for test selection, and for test prioritization [6][8][17][31]. Techniques used to compute program change are source code differencing [6][8][28], data and control flow analysis [1][20], and coarse grained modified code entities [5] to identify which parts of the program might be affected by the changes.

Source code differencing techniques can be built using commonly available tools like the Unix diff [6][8][28], which are simple and fast. This technique will erroneously mark a procedure as changed when a variable in a procedure was simply renamed. It will also fail to determine the set of affected procedures when definitions in a header file, such as macro definitions and method definitions, were modified. Static analysis will be needed to address these cases.

Flow analysis is difficult in a language such as C++/C which contains pointers, casts, and aliasing. Flow analysis is expensive on large commercial programs [9][21]. Graves [9] argues that cost of flow techniques can only be justified if the results were used for other analyses.

The effectiveness of techniques is also dependent on the granularity at which the program change is determined. Harrold [11] shows techniques that determine change at coarser-grained entities like functions [5] may select more tests than statement or control flow based techniques [1][20]. Modification-based prioritization techniques [6][8] compute program changes at the function granularity using the UNIX diff tool.

Echelon uses a binary code based approach for test prioritization. Working at the binary level has advantages over working at the source code level. As binary modification eliminates the recompilation step for collecting coverage etc., it is easier to integrate into the build process in production environments. Moreover, by the time the program is available in binary form, all the changes in header files to constants, macro definitions etc. have already been propagated to the affected procedures in the program, thus simplifying the process for determining program changes. Although program has been compiled to machine code in the binary form, interprocedural flow analysis and optimization can be performed, if needed, at binary level [24][25][26].

Echelon uses binary matching [29] to compute program change. It computes program change at basic block granularity. Echelon utilizes a fast and simple heuristic to predict which tests will cover the affected basic blocks. By doing prioritization, Echelon can use a non-precise algorithm that works well in practice. Echelon's technique is fast and scales well to large programs making it suitable for use in development environments.

3. ARCHITECTURE OF ECHELON

Echelon is part of the Magellan test effectiveness tool set. Echelon leverages the Magellan and Vulcan [23] infrastructure for information and analysis. In this section, we first briefly describe Magellan and Vulcan, and then discuss Echelon in greater detail.

The Magellan Test Effectiveness Infrastructure

The Magellan tool set provides an infrastructure for collecting, storing, analyzing, and reporting information about a test process. The core of Magellan is a SQL Server-based repository that stores test coverage information for each test. The coverage information can be mapped to the static structure of the program: the procedures, files, directories, binaries etc. that make up the program. All the program binaries that were tested and their corresponding symbol files for all relevant versions of the program are stored in a separate symbol repository. The information in the symbol repository can be related to the information in the coverage repository. The Magellan infrastructure is designed to be extensible; Magellan provides a well-defined interface for accessing and storing information.

Although new tools are easily added, Magellan provides a set of tools that are commonly needed during the test process. The toolset includes a test coverage collection tool that uses binary instrumentation to collect block coverage and arc coverage information both in user and kernel mode. The coverage information is collected for each test and stored in Magellan's repository. For easy presentation of coverage data, Magellan provides reporting tools with graphical user interface that can map the data to the source code and Blender, a test migration tool, to migrate coverage data from an older version of the program to the new version. Blender addresses a common need when a new version of the program arrives while testing for the previous version is still in progress. Blender provides a convenient way to migrate testing to the new version without losing what has already been tested, if it is still relevant to the new version.

Binary Modification Infrastructure

Echelon utilizes a rich binary modification infrastructure called Vulcan [23]. Vulcan is a second-generation technology that provides both static and dynamic binary code modification and provides a framework for analysis and optimization. Vulcan provides a uniform abstraction to binary modification with a simple API for inspection, instrumentation and optimization. Vulcan works in the Win32 environment and can process x86, IA64, and MSIL binaries. Vulcan has been used to improve the performance and reliability of many Microsoft products.

To compute the changes between two version of the program, Echelon utilizes BMAT[29], a binary matching tool built using Vulcan. BMAT is a fast and effective tool that matches two versions of a binary program without the knowledge of source code changes. This tool uses a hashing-based matching algorithm and a series of heuristic methods, with the goal of matching as much of the program as possible with high accuracy. The algorithm first matches procedures, then blocks within each procedure. Several levels of matching are attempted with varying degrees of fuzziness. This process allows correct matches to be found even with shifted addresses, different register allocation, and small program modifications. The success rate of matching of code blocks is often higher than 99% [29]. Given two versions of the program, BMAT finds the matches between the old basic blocks and the new basic blocks.

3.1 Echelon: Test Prioritization System

Echelon takes as input two versions of a program in binary form along with the test coverage information of the old binary and produces a prioritized list of the tests, as well as a list of modified and new blocks (or source lines) that may not be executed by any existing test. Echelon accomplishes this task in three steps illustrated in Figure 1.

In the first step, Echelon uses BMAT to find a matching block in the old binary for each block in the new binary. Blocks with no matches are marked as *new blocks*. Blocks with matches are further compared to see if they are identical. (This comparison is done at a logical level as Vulcan's representation is symbolic and contains no hard coded addresses.) Identical blocks are marked as *old blocks*; otherwise, they are marked as *old modified blocks*. For our analysis, we define *impacted blocks* to be the set of old modified blocks and new blocks; these are the blocks that have been changed between the two versions of the program.

In the second step, Echelon tries to determine which impacted blocks in the new version are likely to be covered by an existing test. For the old modified blocks, we simply check to see if the test covered the matching block in the old binary using the coverage information of the old binary. For new blocks, Echelon uses the following simple heuristic.

As Vulcan provides the representation of the program as an inter-procedural graph, all the successor and predecessor blocks for each new block are easily computed. We use the heuristic that a test may cover a new block if it covers at least one of its immediate predecessor blocks and at least one of its immediate successor blocks, skipping in both cases any intermediate new blocks.

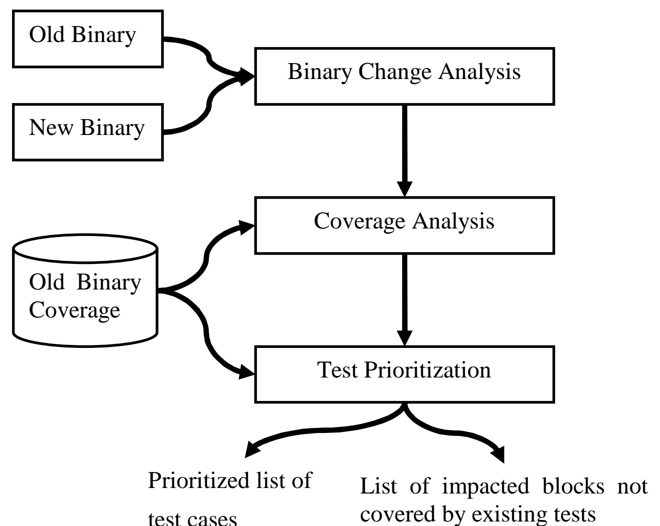

Figure 1. The Echelon system

This heuristic works very well in practice (as shown in Section 5), but it misses some infrequent cases. First, it will miss if there was a path from a predecessor to a successor that did not go through the new block. Echelon can overcome this by using arc coverage or branch prediction [2] to decide which arc is likely to be taken. Second, if a new block was a target of an indirect call, the predecessors of the new block are not always visible in the static graph, causing the heuristic to always predict that these blocks are

not covered by any test. To handle these blocks, Echelon provides an option to relax the condition that requires a predecessor block to be executed; therefore, a test will cover a new block that is a target of an indirect call if it covers at least one of its successor blocks. In both cases, these approaches improve the heuristic.

After these two steps, Echelon has determined the set of impacted blocks (new and old modified blocks) that will be covered by each test.

```
Input:
TestList: set of tests
Coverage (t) : set of blocks covered by  test t
ImpactedBlkSet: set of new and old modified blocks

Output: a set of sequences Seq

Algorithm:
while (any t in TestList covers any block in ImpactedBlkSet )
{
   CurrBlkSet = ImpactedBlkSet
    Start a new sequence Seq
   while (any t in TestList covers any block in CurrBlkSet }
    {
       for each t in TestList  compute
       {
            Weight(t) = count[CurrBlkSet ∩ Coverage(t)]
       }
       Select test  t in TestList with maximum weight
       Add t to current sequence Seq
       Remove t from TestList
       CurrBlkSet = CurrBlkSet – Coverage (p)
      }
}
Put all remaining tests in TestList in a new sequence Seq
```

Figure 2. Ordering tests

In the third step, similar to [6][8][31], Echelon prioritizes the given set of tests. As shown in Figure 2, Echelon uses the impacted block set for each test to prioritize the tests. Echelon uses an iterative, greedy algorithm to first find a short sequence of tests from the given tests such that as many of the impacted blocks as possible will be covered. The algorithm starts by assigning priority weight to each test equal to the number of impacted blocks it covers. The test with the maximum weight is first selected; in case of a tie we pick the test with the maximum overall coverage. The selected test is removed from the list, and the impacted blocks covered by it are removed from the impacted block set, CurrBlkSet. The priority weight for each test is recalculated based on the updated impacted block set, CurrBlkSet. By doing so, the algorithm tries to pick a test from the remaining tests that will cover the maximum number of the remaining impacted blocks. This is repeated till any remaining test can cover any remaining impacted blocks in CurrBlkSet. The tests thus selected form the first sequence that will provide the maximum coverage of the impacted blocks. This process is repeated to generate the next sequence, till any remaining test can cover any of the impacted blocks in ImpactedBlkSet. Remaining tests are added to a separate sequence in the order of their overall coverage. As we keep a sorted list of tests by weight, we terminate the search for a test when the new computed weight is greater than the original weight of the next test. This helps the algorithm to converge faster.

Extension to Prioritization Algorithm

We have described the algorithm using only the coverage of the impacted blocks to compute priority weight. An obvious addition is taking the execution time of the tests into consideration. If two tests provide the same coverage of impacted blocks, one with the shorter time should be selected. Echelon provides an option to take time into consideration and uses rate of impacted block coverage for prioritization; rest of the prioritizing algorithm remains unchanged. Other factors like overall coverage, rate of fault detection etc. can be easily added in computing the priority weight that have been used in previous work [6][8].

4. PERFORMANCE OF ECHELON

Echelon has been implemented in C++ on the Windows environment on the Intel x86 platform. For use in production environments, the performance of Echelon is critical. We measured Echelon's performance on a large production binary, shown in Table 1, on a PIII 993 MHz, Dual proc, 1 GB RAM.

Table 1. Program information

	Version 1	Version 2
Date	December 2000	January 2001
Functions	31,020	31,026
Blocks	668,068	668,274
File Size (bytes)	8,880,128	8,880,128
PDB Size (bytes)	22,602,752	22,651,904
No. of Tests	3128	3128

Table 1 shows two versions of a large office productivity application of 1.8 million lines of source code resulting in an 8.8 Mb executable with 22Mb symbol table. We selected two versions of the binary about a month apart late in the product development cycle. The changes between the two versions are therefore small, thus better exercising the accuracy and effectiveness of Echelon. Table 2 shows only 378 blocks out of the program's 668068 blocks were impacted; 220 new blocks were added while 158 old blocks were modified. The application was run with 3128 tests which take over 4 days to run on a single machine.

Table 2. Program change information

Impacted Blocks	378 (220 New, 158 old)
Impacted Blocks covered	176 Blocks
Tests in first sequence	16 Tests
Number of sequences	1,225
Time taken by Echelon	210 seconds

Echelon took 210 seconds to prioritize the 3128 tests: 90 seconds to compute the program changes using binary matching and 120 seconds to compute the impacted block set and prioritizing the tests.

5. ANALYSIS OF ECHELON RESULTS

Echelon has been tested on a number of binaries from Microsoft's development environment. To understand how well Echelon performs, two measurements are of interest:

- First, how many sequences of tests were formed and how many tests are in each sequence?
- Second, how accurate is Echelon?

We will address these questions in detail on the application binary shown in Table 1.

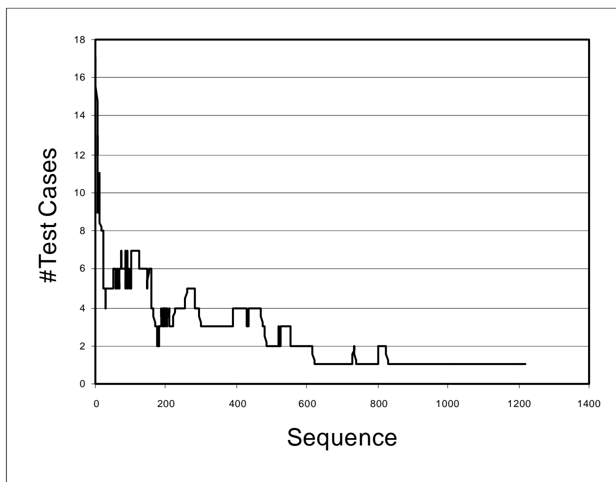

Figure 3. Number of tests in each sequence

Echelon prioritized the 3128 tests into 1225 sequences. The first sequence, which provides a maximum coverage of impacted blocks, contained only 16 tests. Figure 3 shows the number of tests in each sequence. The number of tests in each sequence falls sharply after the first few sequences because most of the changes are concentrated in one part of the program. The graph also shows that about half of the sequences towards the end contain only one test; all these tests cover a common routine that was modified but did not cover much of the other modified code. Echelon has correctly prioritized these tests by putting them towards the end of the list.

Figure 4 shows how many impacted blocks are covered by each minimal sequence. As expected the first sequence covers the maximum number possible. However, the graph shows that there is a sharp decline after the first few sequences. This information in the graph is very useful for the test team while deciding on how many test sequences to run. Interestingly, the sequences towards the end cover the same one block. These correspond to same sequences consisting of single tests in Figure 3.

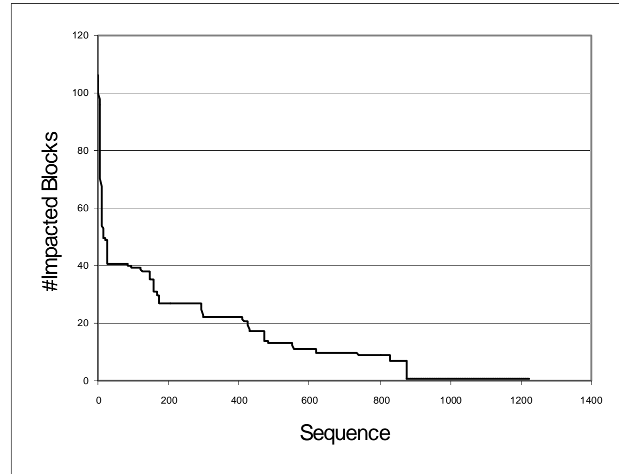

Figure 4. Number of impacted blocks in each sequence

Figure 5 again shows the percentage coverage of impacted blocks attained by each sequence as well as the cumulative overall coverage attained as we proceed down the list of sequences. The impacted block coverage is the highest for the first sequence and it decreases, as expected, as we go along. Although Echelon does not take overall coverage into consideration except to break ties, the overall coverage reaches the maximum at a very fast pace. (Note that some impacted blocks were not covered by any existing test.) This information is also useful in deciding how many sequences to run when running in time constrained environments.

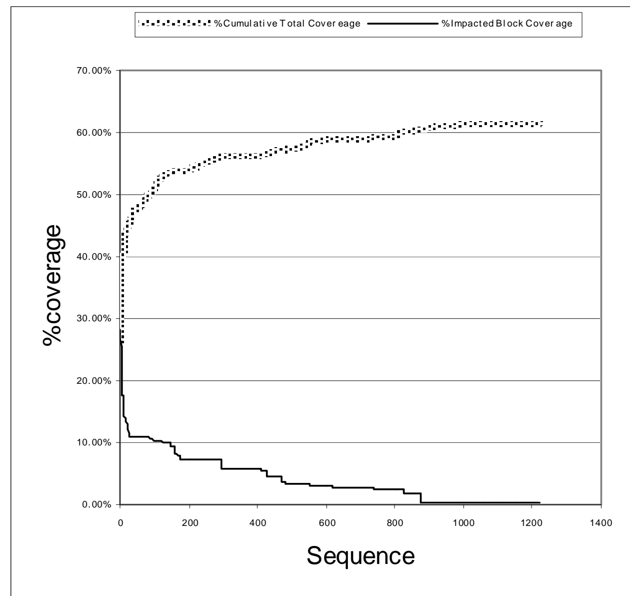

Figure 5. Cumulative coverage and impacted coverage

To measure Echelon's accuracy, we looked at two questions.

- First, how many blocks that were predicted to be covered by a test were not in fact covered?
- Second, how many blocks that were predicted to be not covered by a test were actually covered?

To compute this information, we collected full coverage information for the two versions and used that to verify how well Echelon predicted. (This collection of coverage information is not part of normal operation of Echelon.)

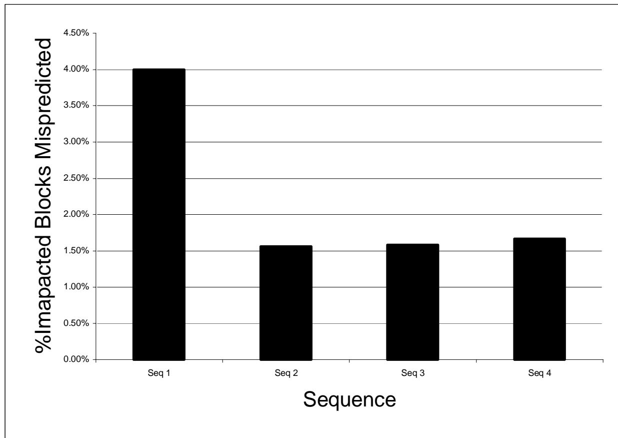

Figure 6. Predicted blocks not covered

Figure 6 shows the percentage of impacted blocks in the first four sequences that were predicted to be covered by a test but were in fact not covered. The error was in the range of 1-4%. In each of these cases, Echelon did not predict correctly because there was also a direct path from the predecessor to the successor of a new block. We could have avoided the error if we had used the available arc coverage information to predict which edges are likely to execute.

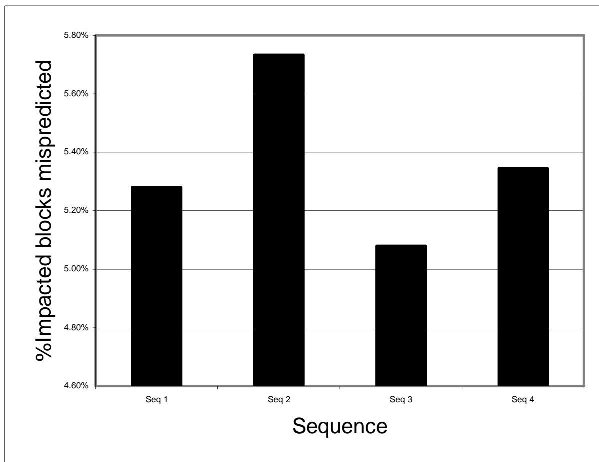

Figure 7. Blocks covered but not predicted

Figure 7 shows the percentage of impacted blocks in the first four sequences that were predicted not to be covered by any test but actually were. This error was in the range of 5-6%. In these cases, a number of new blocks were inserted at the head of a procedure. As the procedure was called indirectly, no predecessor of these new block were visible in the graph. As we did not use the option to compensate for indirect calls, Echelon incorrectly predicted that these blocks would not be covered.

Below are results on two binaries from a Microsoft Windows operating system. They are consistent with our previous results. This data was collected using a partial set of tests. The tests are coarse; each test comprises many individual tests.

Table 3. Program and Change Information

	Version 1	Version 2
Date	05/01/2001	05/23/2001
Functions	1,761	1,774
Blocks	32,012	32,135
Arcs	47,131	47,323
File size (bytes)	882,688	894,464
Impacted Blocks	0	589 (350 N, 239 OC)
Tests	56	56
Time taken to prioritize	-	29 seconds

Table 3 shows two versions of an operating system binary. Only 589 blocks out of the program's 32135 blocks were impacted. Echelon divided the tests into 27 sequences with the first sequence containing only 3 tests as shown in Figure 8.

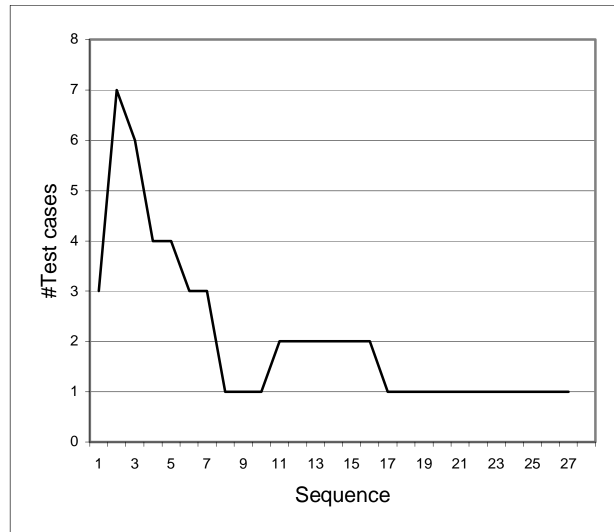

Figure 8. Number of tests in each sequence

Figure 9 shows 176 impacted blocks were covered by the first sequence.

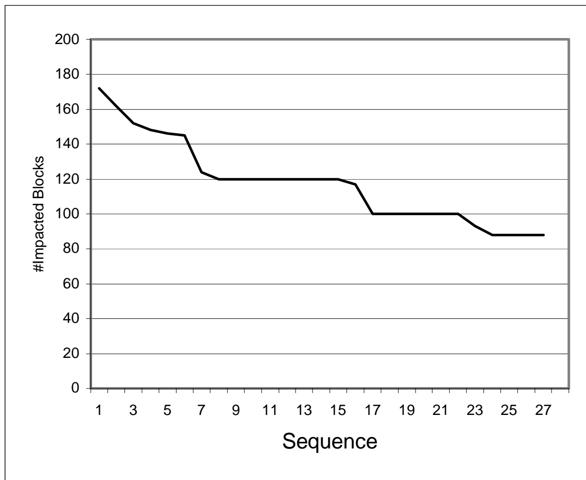

Figure 9. Number of impacted blocks in each sequence

Table 4 shows two versions of another operating system binary. Only 270 blocks out of the program's 31003 blocks were impacted.

Table 4. Program and change information

	Version 1	Version 2
Date	05/01/2001	05/23/2001
Functions	1,967	1,970
Blocks	30,916	31,003
Arcs	46,638	46,775
File size (bytes)	528,384	528,896
Impacted Blocks	0	270 (190 N, 80 OC)
Tests	56	56
Time taken		21 seconds

Echelon divided the tests into 34 sequences with the first sequence containing only 1 test as shown in Figure 10.

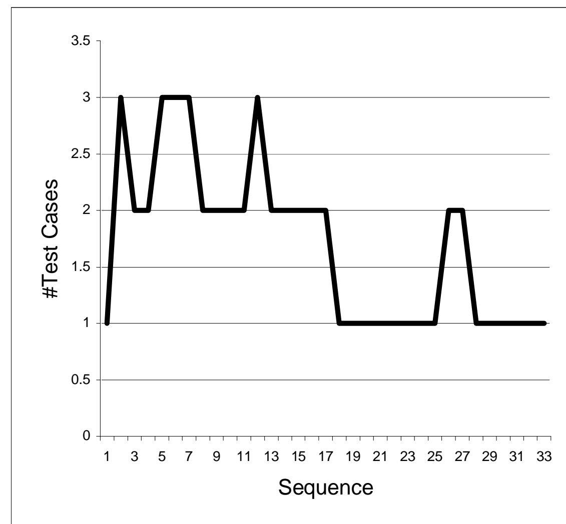

Figure 10. Number of tests in each sequence

Figure 11 shows 66 impacted blocks were covered by the first sequence.

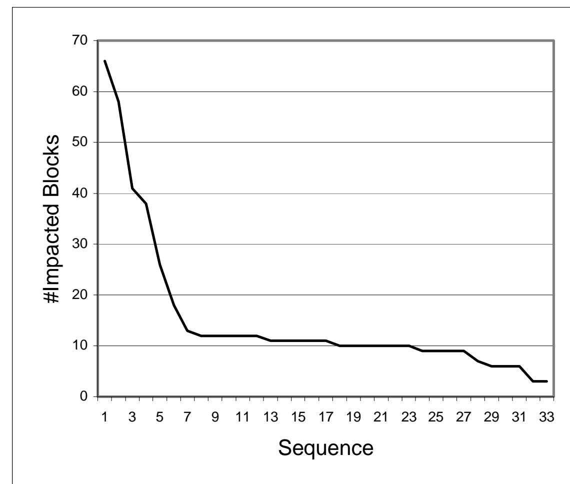

Figure 11. Number of impacted blocks in each sequence

6. EFFECTIVENESS OF ECHELON

An important measure of effectiveness of a prioritization technique is the early detection of defects by running the tests in the specified order. A number of empirical studies to measure various test prioritization techniques have been conducted in [6][7][8][18][31]. To measure the effectiveness of Echelon, we studied the defect detection on binaries from a development process; the preliminary results on two binaries are discussed below.

Echelon prioritized 157 tests for a binary from the development process for which the coverage data was available. The binary

(size = 410KB, functions = 2308, blocks = 22446) had a known number of defects. Echelon divided the tests into 148 sequences (sequence 1 – 6 tests, sequence 2 – 4 tests, sequence 3 – 2 tests, sequence 4 – 1 test). Figure 12 shows the percentage of defects detected by the first four sequences. It also shows the percentage of unique defects, which are defects that were not detected by any of the previous sequences. As Figure 12 shows, the first sequence detected 81% of the defects. The second sequence detected 62% of the defects; 6% of these defects were unique while 56% of the defects had already been detected by the first sequence. The rest of the sequences did not detect any unique defects. None of the existing tests detected the remaining 13% of the defects; new tests will be needed to detect them.

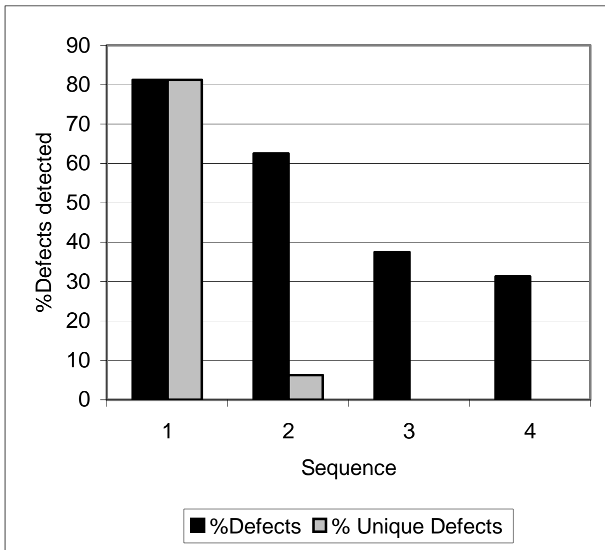

Figure 12. Defects detected in each sequence

We conducted a similar study for another binary from the development process. Echelon prioritized 221 tests for two versions of a binary (size = 400KB, functions = 3675, blocks = 23153) from the development process which had a known set of defects. Echelon divided the tests into 176 sequences (sequence 1 – 4 tests, sequence 2 – 4 tests, sequence 3 – 3 tests, sequence 4 – 5 tests). As Figure 13 shows, the first sequence detected 85% of the defects. The second sequence detected 71% of the defects; 9% of these defects were unique while 62% of the defects had already been detected by the first sequence. The third sequence detected 57% of the defects; 4% of these defects were unique while 53% of the defects had already been detected by the previous sequences. The rest of the sequences did not detect any unique defects. None of the existing tests detected the remaining 2% of the defects; new tests will be needed to detect them.

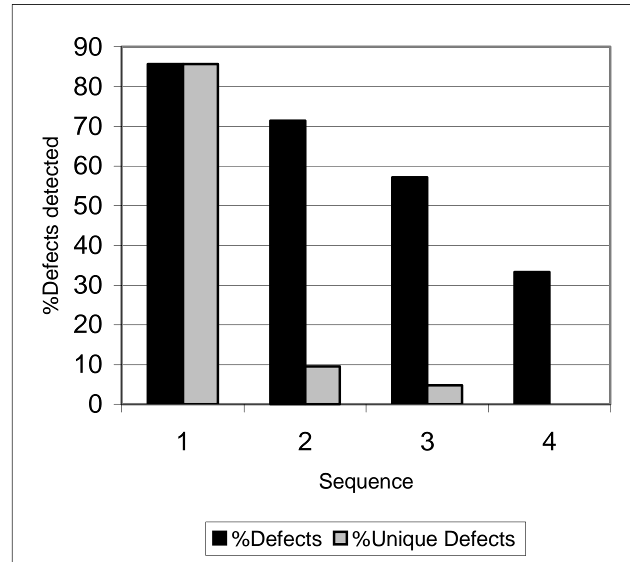

Figure 13. Defects detected in each sequence

7. FUTURE WORK

This technology can be applied effectively to address other testing strategies, for example, "real-time" testing: test teams, for example, have performance regression test suites that verify if program changes have not adversely affected program performance. Test teams are often faced with the question of how often or when to run these performance regression suites. Through performance measurement tools, we already know parts of the program where the application spends most of its time on critical programs. Echelon can accurately determine if any performance critical parts of the program have been changed and then assign additional appropriate weight to the performance regression tests leading them to be prioritized higher. A natural extension is to use more information like performance from the development process to guide testing decisions.

Also, as we collect more data from the development environment, we will track the effectiveness of Echelon in finding defects.

8. CONCLUSIONS

The paper shows that it is possible to effectively prioritize tests in large scale software development environments. By using binary matching to determine changes at a fine granularity along with a simple and intuitive algorithm, Echelon can prioritize tests based on changes between two versions of the program. Echelon is able to operate on large binaries built from millions of lines of source code and produce results within a few minutes. This approach enables effective testing early in the development process that saves time and resources. Echelon is being integrated into the Microsoft development process; we'll continue to refine and extend the algorithms as we learn more from its usage in real production environments.

9. ACKNOWLEDGMENTS

Many people have helped bring Echelon to its current stage. The Magellan and Vulcan team worked tirelessly to integrate and deploy this work in Microsoft's development environment. Our debugging and verification of results greatly benefited from Andy Edwards' visualization tool that enabled us to visualize Vulcan's representation. Loren Merriman helped us in getting the initial measurements. Pankaj Lunia and Norberto Arrieta were the early adopters and braved through the early versions of the system. John DeTreville, G.S. Rana, David Notkin, Jim Larus, Tom Ball, and the anonymous referees provided perceptive comments on the early drafts of the paper. Our sincere thanks to all.

REFERENCES

[1] T. Ball, "On the Limit of Control Flow Analysis for Regression Test Selection". Proc. ACM Int'l Symposium. Software Testing and Analysis, pp. 134-142, Mar. 1998.

[2] T. Ball and J. Larus, Branch Prediction for Free, Proceedings of Programming Language Design and Implementation, 1993.

[3] D. Binkley, "Semantics guided Regression Test Cost Reduction", IEEE Trans. Software Eng., vol. 23, no. 8, pp. 498-516, Aug. 1997.

[4] T.Y. Chen and M.F. Lau, "Dividing Strategies for the Optimization of a Test Suite", Information Processing Letters, vol. 60, no. 3, pp. 135-141, Mar. 1996.

[5] Y.F. Chen, D.S. Rosenblum, and K.P. Vo, "TestTube: A System for Selective Regression Testing," Proc. 16th Int'l Conf. Software Eng., pp. 211-222, May 1994.

[6] S. Elbaum, .A. Malishevsky and G. Rothermel, "Test case prioritization: A family of empirical studies", IEEE Trans. Software Engg. , vol. 28, no. 2, pp. 159-182, Feb. 2002.

[7] S. Elbaum, .A. Malishevsky and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization", Proc. 23rd Int'l Conf. Software Engg., pp. 329-338, May 2001.

[8] S. Elbaum, .A. Malishevsky and G. Rothermel, "Prioritizing test cases for regression testing", Proc. Int'l Symp. Software Testing and Analysis, pp. 102-112, Aug. 2000.

[9] T. L. Graves, M.J. Harrold, J-M. Kim, A. Porter and G. Rothermel, "An empirical study of regression test selection techniques", 20th Int'l Conference on Software Engineering, Apr. 1998.

[10] M.J. Harrold and G. Rothermel, "Empirical Studies of a Prediction Model for Regression Test Selection", IEEE Trans. On Software Eng., vol. 27, no. 3, Mar. 2001.

[11] M. J. Harrold, "Testing Evolving Software", Journal of Systems and Software, vol. 47, no. 2-3, pp. 173-181, Jul. 1999.

[12] M.J. Harold and G. Rothermel, "Aristotle: A System for Research on and Development of Program Analysis Based Tools", Technical Report OSU-CISRC-3/97-TR17, The Ohio State Univ., Mar. 1997.

[13] M.J. Harrold, R. Gupta and M.L. Soffa, "A Methodology for Controlling the Size of a Test Suite", ACM Trans. Software Eng. And Methodology, vol. 2, no. 3, pp. 270-285, July 1993.

[14] J.R. Horgan, and S.A. London, "ATAC: A data flow coverage testing tool for C", Proc. of Symp. On Assessment of Quality Software Development Tools, pp. 2-10, 1992.

[15] D. Rosenblum and G. Rothermel, "An Empirical comparison of regression test selection techniques", Proceedings of the Int'l Workshop for Empirical Studies of Software Maintenance, Oct. 1997.

[16] D.S. Rosenblum and E.J. Weyuker, "Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies", IEEE Trans. Software Engineering, vol. 23, no. 3, pp. 146-156, Mar. 1997.

[17] G. Rothermel, R.H. Untch and M.J. Harrold, "Prioritizing Test Cases For Regression Testing", IEEE trans. On Software Engineering, vol. 27, no. 10, Oct. 2001

[18] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, "Test Case Prioritization: An Empirical Study", Proc. Int'l Conf. Software Maintenance, pp. 179-188, Aug. 1999.

[19] G. Rothermel, M.J. Harrold, J. Ostrin and C. Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites", Proc. Int'l Conf. Software Maintenance, pp. 34-43, Nov. 1998.

[20] G. Rothermel and M.J. Harrold, "A Safe, Efficient Regression Test Selection Technique", ACM Trans. Software Eng. And Methodology, vol. 6, no. 2, pp. 173-210, Apr. 1997.

[21] G. Rothermel and M. J. Harrold, "Experience with Regression Test Selection", Proc. of the Int'l Workshop for Empirical Studies of Software Maintenance, Monterrey, CA, Nov. 1996.

[22] G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques", IEEE Trans. Software Eng., vol. 22, no. 8, pp. 529-551, Aug. 1996.

[23] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary Transformation in a Distributed Environment", Microsoft Research Technical Report, MSR-TR-2001-50.

[24] A. Srivastava and D. Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. Symposium on Programming Language Design and Implementation, 1994, pp 49-60.

[25] A. Srivastava and A. Eustace, "ATOM – A System for Building Customized Program Analysis Tools", Symposium on Programming Language Design and Implementation, 1994, pp. 196-205, 1994.

[26] A. Srivastava and D. Wall. A Practical System for Intermodule Code Optimization at Link Time. Journal of Programming Language, 1(1):1-18, March 93.

[27] F. Vokolos and P. Frankl, "Empirical evaluation of the textual differencing regression testing techniques", Int'l conference on Software Maintenance, Nov. 1998.

[28] F. Vokolos and P. Frankl, “Pythia: a regression test selection tool based on text differencing”, Int’l conference on reliability, Quality and Safety of Software Intensive Systems, May 1997.

[29] Z. Wang, K. Pierce, and S. McFarling, “BMAT: A Binary Matching Tool for Stale Profile Propagation”, The Journal of Instruction-Level Parallelism, vol. 2, May 2000.

[30] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur, “Effect of Test Set Minimization on Fault Detection Effectiveness”, Software-Practice and Experience, vol. 28. no. 4, pp. 347-369, Apr. 1998.

[31] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal, “A Study of Effective Regression Testing in Practice”, Proc. Eighth Int’l Symposium Software Reliability Eng., pp. 230-238, Nov. 1997.

[32] W.E. Wong, J.R. Horgan, A.P. Mathur, and A. Pasquini, “Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application”, Proc. 21st Ann. Int’l Computer Software & Applications Conf., pp. 522-528, Aug. 1997.

.