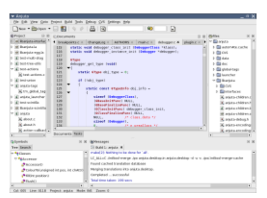
PLDI 2005 -- Ben's slides!

Scalable Statistical Bug Isolation

Ben Liblit, *Mayur Naik, Alice Zheng, Alex Aiken, and Michael Jordan*

University of Wisconsin, Stanford University, and UC Berkeley

Post-Deployment Monitoring



Goal: Measure Reality

- Where is the black box for software?
  - Crash reporting systems are a start
- Actual runs are a vast resource
  - Number of real runs >> number of testing runs
  - Real-world executions are most important
- This talk: post-deployment bug hunting
  - Mining feedback data for causes of failure

What Should We Measure?

- Function return values
- Control flow decisions
- Minima & maxima
- Value relationships
- Pointer regions
- Reference counts
- Temporal relationships

```
err = fetch(file, &obj);
if (!err && count < size)
    list[count++] = obj;
else
    unref(obj);
```

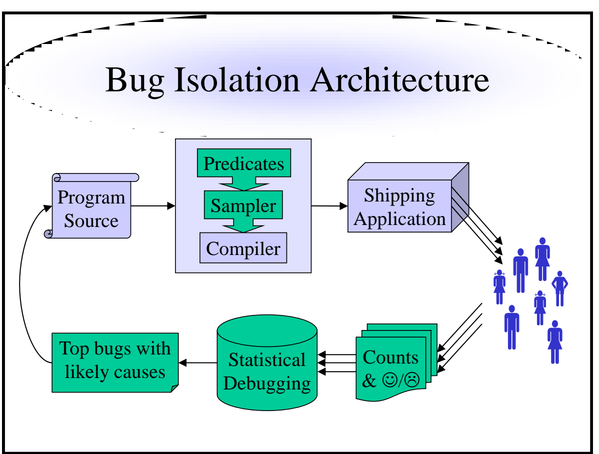
In other words, *lots* of things

Our Model of Behavior

Any interesting behavior is expressible as a predicate P on program state at a particular program point.

Count how often "P observed true" and "P observed" using sparse but fair random samples of complete behavior.

Bug Isolation Architecture



Find Causes of Bugs

- Gather information about *many* predicates
  - 298,482 predicates in `bc`
  - 857,384 predicates in `Rhythmbox`
- Most are not predictive of anything
- How do we find the useful bug predictors?
  - Data is incomplete, noisy, irreproducible, …

Look For Statistical Trends

How likely is failure when P happens?

F(P) = # of failures where P observed true
S(P) = # of successes where P observed true

$$\text{Failure}(P) = \frac{F(P)}{F(P) + S(P)}$$

Good Start, But Not Enough

```
if (f == NULL) {
  x = 0;
  *f;
}
```

Failure(f == NULL) = 1.0

Failure(x == 0) = 1.0

- Predicate x == 0 is an innocent bystander
  - Program is already doomed

Context

What is the background chance of failure regardless of P's truth or falsehood?

F(P observed) = # of failures observing P
S(P observed) = # of successes observing P

$$\text{Context(P)} = \frac{\text{F(P observed)}}{\text{F(P observed)} + \text{S(P observed)}}$$

Isolate the Predictive Value of P

Does P being true *increase* the chance of failure over the background rate?

Increase(P) = Failure(P) – Context(P)

(a form of likelihood ratio testing)

Increase() Isolates the Predictor

```
if (f == NULL) {
  x = 0;
  *f;
}
```

Increase(f == NULL) = 1.0

Increase(x == 0) = 0.0

Isolating a Single Bug in `bc`

```
void more_arrays ()
{
  …

  /* Copy the old arrays. */
  for (indx = 1; indx < old_count
    arrays[indx] = old_ary[indx];

  /* Initialize the new elements. */
  for (; indx < v_count; indx++)
    arrays[indx] = NULL;

  …
}
```

#1: `indx > scale`
#2: `indx > use_math`
#3: `indx > opterr`
#4: `indx > next_func`
#5: `indx > i_base`

It Works!

…for programs with just one bug.

- Need to deal with multiple, unknown bugs
- Redundant predictors are a major problem

Goal: Isolate the best predictor for each bug, with no prior knowledge of the number of bugs.

Multiple Bugs: Some Issues

- A bug may have many redundant predictors
  - Only need one, provided it is a good one
- Bugs occur on vastly different scales
  - Predictors for common bugs may dominate, hiding predictors of less common problems

Guide to Visualization

- Multiple interesting & useful predicate metrics
- Graphical representation helps reveal trends

Increase(P)
error bound
Context(P)
S(P)
log(F(P) + S(P))

Bad Idea #1: Rank by Increase(P)

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.065	0.935 ± 0.019	0	23	23	((*(fi + i)))->this.last_token < filesbase
	0.065	0.935 ± 0.020	0	10	10	((*(fi + i)))->other.last_line == last
	0.071	0.929 ± 0.020	0	18	18	((*(fi + i)))->other.last_line == filesbase
	0.073	0.927 ± 0.020	0	10	10	((*(fi + i)))->other.last_line == yy_n_chars
	0.071	0.929 ± 0.028	0	19	19	bytes <= filesbase
	0.075	0.925 ± 0.022	0	14	14	((*(fi + i)))->other.first_line == 2
	0.076	0.924 ± 0.022	0	12	12	((*(fi + i)))->this.first_line < nid
	0.077	0.923 ± 0.023	0	10	10	((*(fi + i)))->other.last_line == yy_init
2732 additional predictors follow						

- High Increase() but very few failing runs!
- These are all *sub-bug predictors*
  - Each covers one special case of a larger bug
- Redundancy is clearly a problem

Bad Idea #2: Rank by F(P)

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.007 ± 0.012	22554	5045	27599	files[filesindex].language != 15
	0.176	0.007 ± 0.012	22566	5045	27611	tmp == 0 is FALSE
	0.176	0.007 ± 0.012	22571	5045	27616	strcmp != 0
	0.176	0.007 ± 0.013	18894	4251	23145	tmp == 0 is FALSE
	0.176	0.007 ± 0.013	18885	4240	23125	files[filesindex].language != 14
	0.176	0.008 ± 0.013	17757	4007	21764	filesindex >= 25
	0.177	0.008 ± 0.014	16453	3731	20184	new value of M < old value of M
	0.176	0.261 ± 0.023	4800	3716	8516	config.winnowing_window_size != argc
2732 additional predictors follow						

- Many failing runs but low Increase()!
- Tend to be *super-bug predictors*
  - Each covers several bugs, plus lots of junk

A Helpful Analogy

- In the language of information retrieval
  - Increase(P) has high precision, low recall
  - F(P) has high recall, low precision
- Standard solution:
  - Take the harmonic mean of both
  - Rewards high scores in both dimensions

Rank by Harmonic Mean

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.824 ± 0.009	0	1585	1585	files[filesindex].language > 16
	0.176	0.824 ± 0.009	0	1584	1584	strcmp > 0
	0.176	0.824 ± 0.009	0	1580	1580	strcmp == 0
	0.176	0.824 ± 0.009	0	1577	1577	files[filesindex].language == 17
	0.176	0.824 ± 0.009	0	1576	1576	tmp == 0 is TRUE
	0.176	0.824 ± 0.009	0	1573	1573	strcmp > 0
	0.116	0.883 ± 0.012	1	774	775	((*(fi + i)))->this.last_line == 1
	0.116	0.883 ± 0.012	1	776	777	((*(fi + i)))->other.last_line == yyleng
2732 additional predictors follow						

- It works!
  - Large increase, many failures, few or no successes
- But redundancy is still a problem

Redundancy Elimination

- One predictor for a bug is interesting
  - Additional predictors are a distraction
  - Want to explain each failure once
- Similar to minimum set-cover problem
  - Cover all failed runs with subset of predicates
  - Greedy selection using harmonic ranking

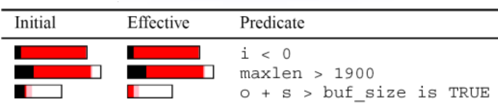
Simulated Iterative Bug Fixing

1. Rank all predicates under consideration
2. Select the top-ranked predicate P
3. Add P to bug predictor list
4. Discard P and all runs where P was true
   - Simulates fixing the bug predicted by P
   - Reduces rank of similar predicates
5. Repeat until out of failures or predicates

Simulated Iterative Bug Fixing

1. Rank all predicates under consideration
2. Select the top-ranked predicate P
3. Add P to bug predictor list
4. Discard P and all runs where P was true
   - Simulates fixing the bug predicted by P
   - Reduces rank of similar predicates
5. Repeat until out of failures or predicates

Experimental Results: `exif`



Initial	Effective	Predicate
		`i < 0`
		`maxlen > 1900`
		`o + s > buf_size is TRUE`

- 3 bug predictors from 156,476 initial predicates
- Each predicate identifies a distinct crashing bug
- All bugs found quickly using analysis results

Experimental Results: Rhythmbox

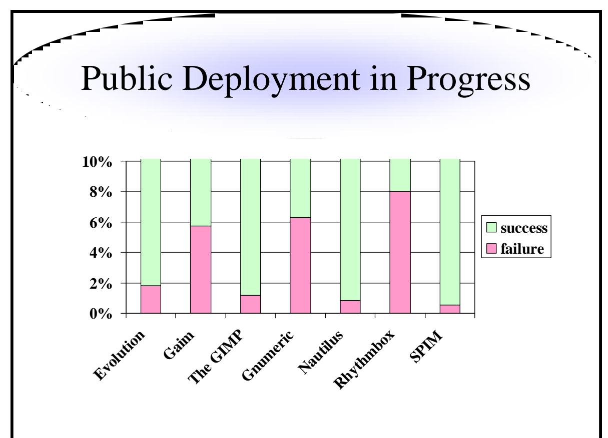
Initial	Effective	Predicate
		tmp is FALSE
		(mp->priv)->timer is FALSE
		(view->priv)->change_sig_queued is TRUE
		(hist->priv)->db is TRUE
		rb_playlist_manager_signals[0] > 269
		(db->priv)->thread_reaper_id >= 12
		entry == entry
		fn == fn
		klass > klass
		genre < artist
		vol <= (float)0 is TRUE
		(player->priv)->handling_error is TRUE
		(statusbar->priv)->library_busy is TRUE
		shell < shell
		len < 270

- 15 bug predictors from 857,384 initial predicates
- Found and fixed several crashing bugs

Lessons Learned

- Can learn a lot from actual executions
  – Users are running buggy code anyway
  – We should capture some of that information
- Crash reporting is a good start, but…
  – Pre-crash behavior can be important
  – Successful runs reveal correct behavior
  – Stack alone is not enough for 50% of bugs

Public Deployment in Progress



Join the Cause!

The Cooperative Bug Isolation Project

http://www.cs.wisc.edu/cbi/

"Borrowed" with only minor reduction – thank you, Amitabh and Jay!

Effectively Prioritizing Tests in Development Environment

ISSTA 2002

Amitabh Srivastava

Jay Thiagarajan
PPRC, Microsoft Research

Using program changes

- Source code differencing
  - S. Elbaum, A. Malishevsky & G. Rothermel "Test case prioritization: A family of empirical studies", Feb. 2002
  - S. Elbaum, A. Malishevsky & G. Rothermel "Prioritizing test cases for regression testing" Aug. 2000
  - F. Vokolos & P. Frankl, "Pythia: a regression test selection tool based on text differencing", May 1997

Using program changes

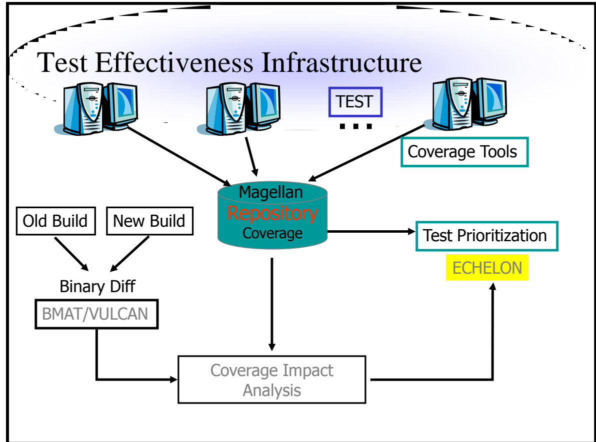
- Data and control flow analysis
  - T. Ball, "On the limit of control flow analysis for regression test selection" Mar. 1998
  - G. Rothermel and M.J. Harrold, "A Safe, Efficient Regression Test Selection Technique" Apr. 1997
- Code entities
  - Y. F. Chen, D.S. Rosenblum and K.P. Vo "TestTube: A System for Selective Regression Testing" May 1994

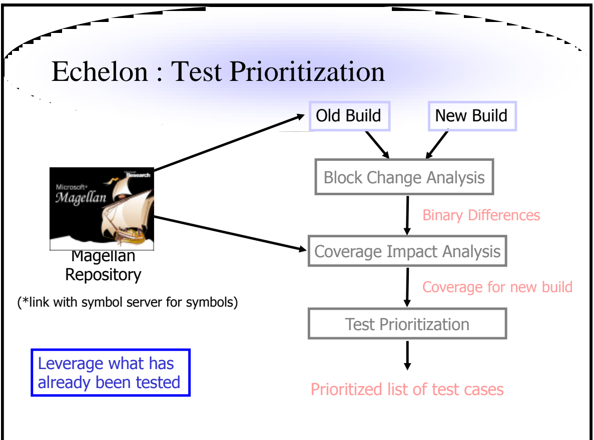
Analysis of various techniques

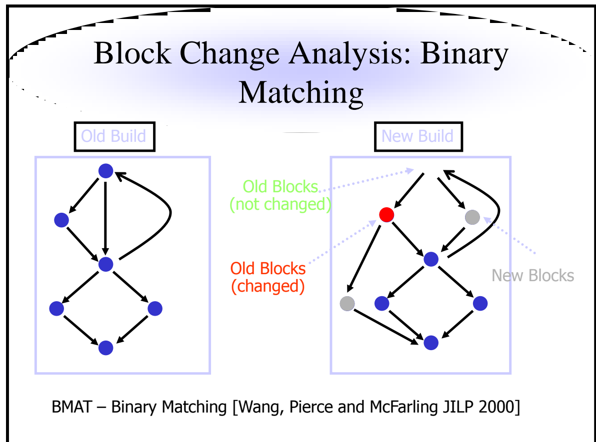
- Source code differencing
  - Simple and fast
  - Can be built using commonly available tools like "diff"
  - Simple renaming of variable will trip off
  - Will fail when macro definition changes
  - To avoid these pitfalls, static analysis is needed
- Data and control flow analysis
  - Flow analysis is difficult in languages like C/C++ with pointers, casts and aliasing
  - Interprocedural data flow techniques are extremely expensive and difficult to implement in complex environment

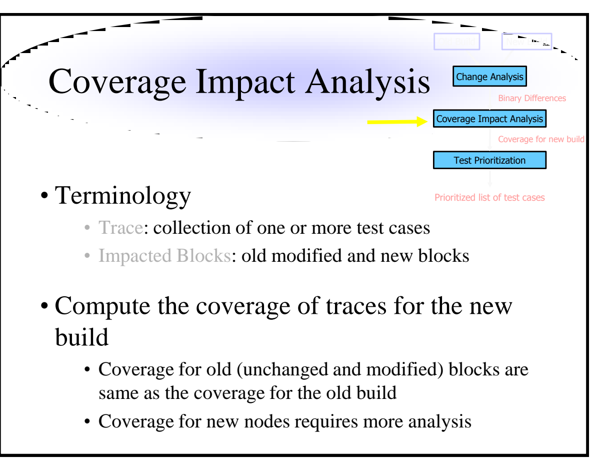
Our Solution

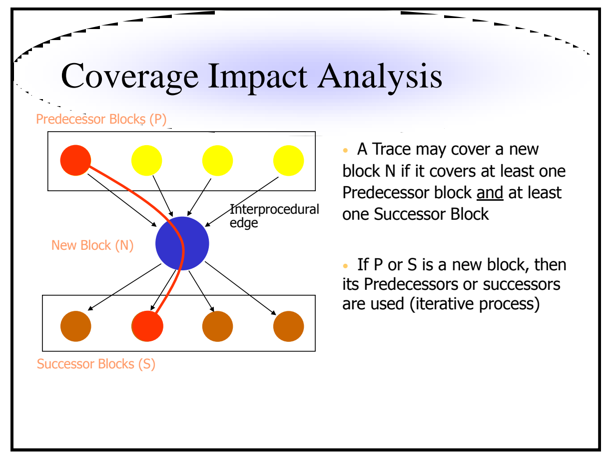
- Focus on change from previous version
  - Determine change at very fine granularity – basic block/instruction
- Operates on binary code
  - Easier to integrate in production environment
  - Scales well to compute results in minutes
- Simple heuristic algorithm to predict which part of code is impacted by the change











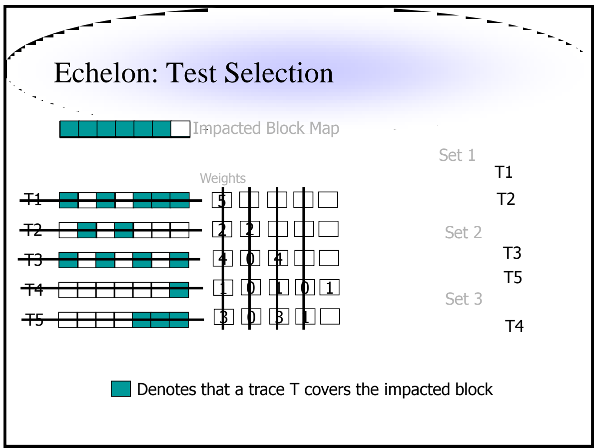
Coverage Impact Analysis

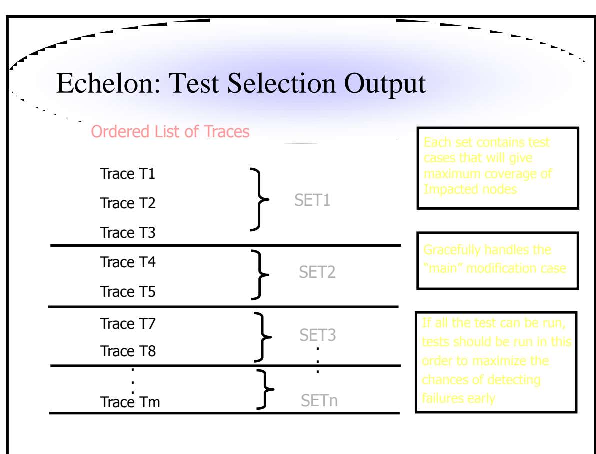
- Limitations - New node may not be executed
  - If there is a path from successor to predecessor
  - If there are changes in control path due to data changes

Echelon : Test Case Prioritization

- Detects minimal sets of test cases that are likely to cover the impacted blocks (old changed and new blocks)
  - Input is traces (test cases) and a set of impacted blocks
  - Uses a greedy iterative algorithm for test selection

Change Analysis
Binary Differences
Coverage Impact Analysis
Coverage for new build
Test Prioritization
Prioritized list of test cases





Analysis of results

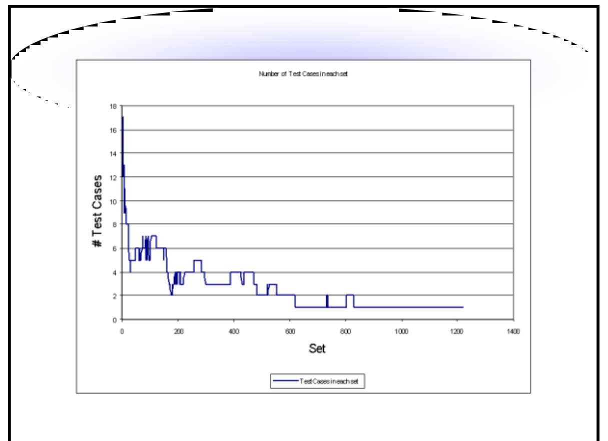
Three measurements of interest

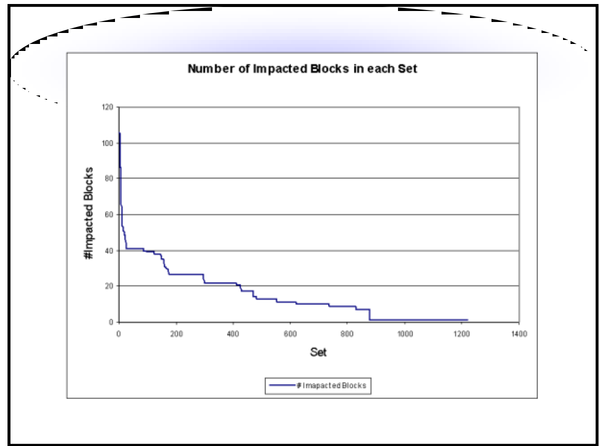
- How many sequences of tests were formed ?
- How effective is the algorithm in practice ?
- How accurate is the algorithm in practice ?

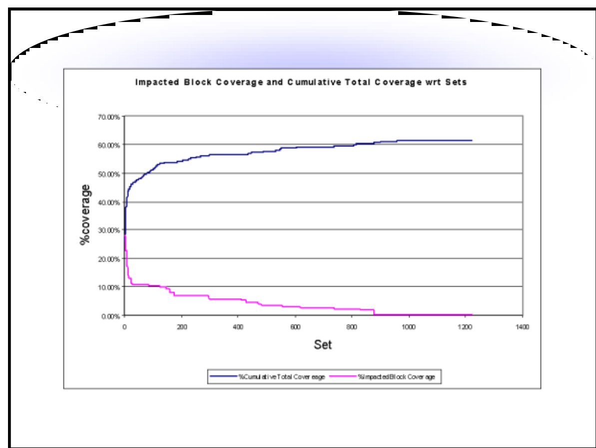
Details of BinaryE

	Version 1	Version 2
Date	12/11/2000	01/29/2001
Functions	31,020	31,026
Blocks	668,068	668,274
Arcs	1,097,294	1,097,650
File size	8,880,128	8,880,128
PDB size	22,602,752	22,651,904
Impacted Blocks	0	378 (220 N, 158 OC)
Number of Traces	3128	3128
# Source Lines	~1.8 Million	~1.8 Million

Echelon takes ~210 seconds for this 8MB binary

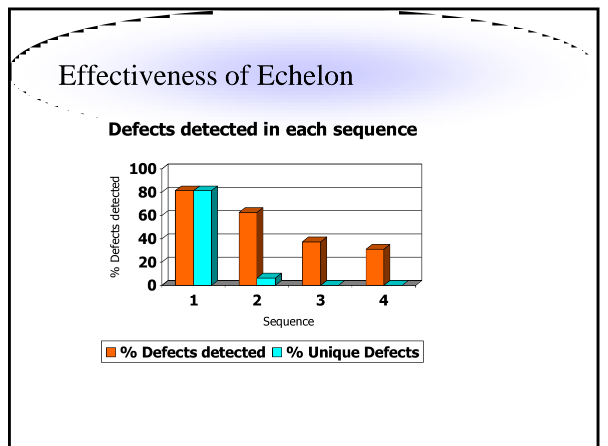


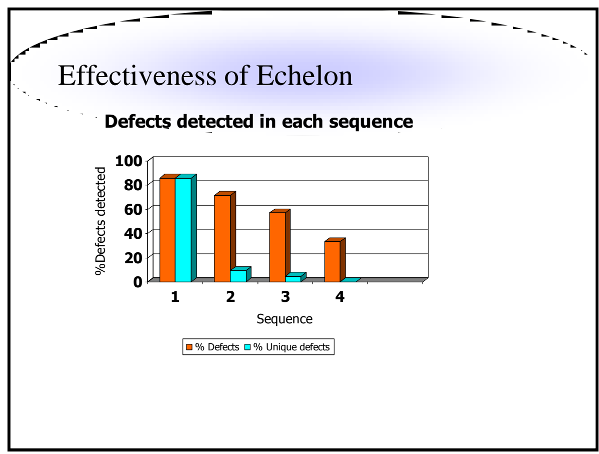


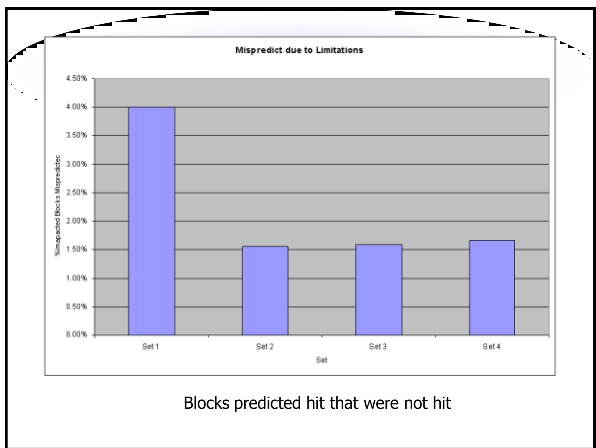


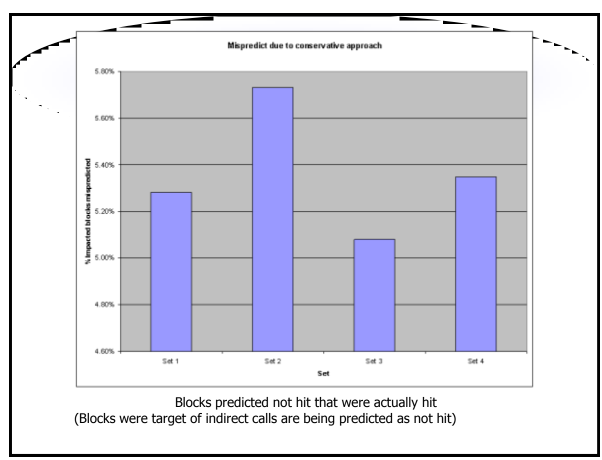
Effectiveness of Echelon

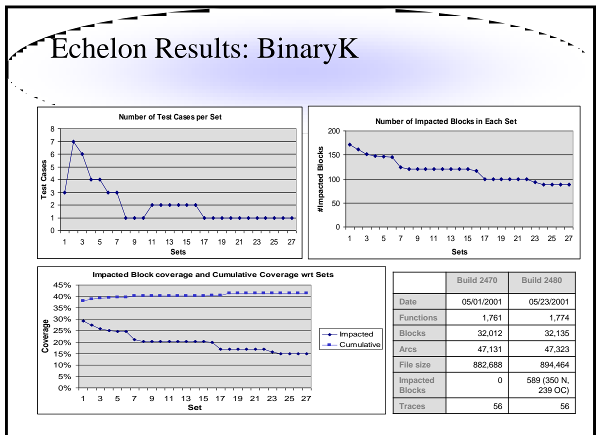
- Important Measure of effectiveness is early defect detection
- Measured % of defects vs. % of unique defects in each sequence
- Unique defects are defects not detected by the previous sequence


Effectiveness of Echelon
Defects detected in each sequence
100
80
60
40
20
0
% Defects detected
1
2
3
4
Sequence
% Defects detected
% Unique Defects

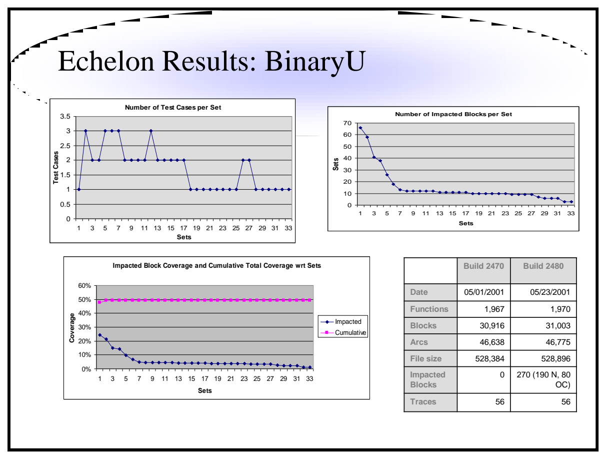

Effectiveness of Echelon
Defects detected in each sequence
100
80
60
40
20
0
%Defects detected
1
2
3
4
Sequence
% Defects
% Unique defects


Mispredict due to Limitations
4.50%
4.00%
3.50%
3.00%
2.50%
2.00%
1.50%
1.00%
0.50%
0.00%
Set 1
Set 2
Set 3
Set 4
Set
Blocks predicted hit that were not hit


Mispredict due to conservative approach
5.80%
5.60%
5.40%
5.20%
5.00%
4.80%
4.60%
Set 1
Set 2
Set 3
Set 4
Set
Blocks predicted not hit that were actually hit
(Blocks were target of indirect calls are being predicted as not hit)



	Build 2470	Build 2480
Date	05/01/2001	05/23/2001
Functions	1,761	1,774
Blocks	32,012	32,135
Arcs	47,131	47,323
File size	882,688	894,464
Impacted Blocks	0	589 (350 N, 239 OC)
Traces	56	56



	Build 2470	Build 2480
Date	05/01/2001	05/23/2001
Functions	1,967	1,970
Blocks	30,916	31,003
Arcs	46,638	46,775
File size	528,384	528,896
Impacted Blocks	0	270 (190 N, 80 OC)
Traces	56	56

Summary

- Binary based test prioritization approach can effectively prioritize tests in large scale development environment
- Simple heuristic with program change in fine granularity works well in practice
- Currently integrated into Microsoft Development process

Coverage Impact Analysis

- Echelon provides a number of options
  - Control branch prediction
  - Indirect calls : if N is target of an indirect call a trace needs to cover at least one of its successor block
- Future improvements include heuristic branch prediction
  – Branch Prediction for Free [Ball, Larus]

Echelon: Test Selection

- Options
  - Calculations of weights can be extended, e.g. traces with great historical fault detection can be given additional weights
  - Include time each test takes into calculation
  - Print changed (modified or new) source code that may not be covered by any trace
  - Print all source code lines that may not be covered by any trace