Dynamic Code Coverage using dyninstAPI

Mustafa Murat Tikir

Outline

- Motivation
- Extension to dyninstAPI
- Using Dominator Tree
- Code Coverage Algorithm
- Current Status
- Experiments and Results
- Conclusion and Contribution

Motivation

- Code Coverage is useful for ensuring bug-free software
  - identifying source code lines not executed in a run or runs
  - making sure that each path is taken at least once during the testing phase
- Useful for identifying bottlenecks in basic block level
- Dynamic approach will produce faster code coverage results for long running programs

Extension to dyninstAPI

- New classes added to dyninstAPI
  - BPatch_basicBlock
  - BPatch_sourceBlock
  - BPatch_flowGraph
- Arbitrary Instrumentation points
  - Conservative base trampoline
- Base trampoline deletion

class BPatch_basicBlock

- machine code basic block
- contains information
  - start/end address of the block in executable
  - outgoing/incoming edges to/from other basic blocks
  - corresponding source code line information
  - immediate dominator and basic blocks dominated immediately
  - delay instruction included if exists
- creation is machine dependent
  - machine specific functions/classes for machine independence

class BPatch_sourceBlock

- source code segment corresponding to a machine basic block
- contains set of corresponding source line numbers
- created one for each machine basic block

© 2000 tikir@cs.umd.edu

Arbitrary instrumentation points

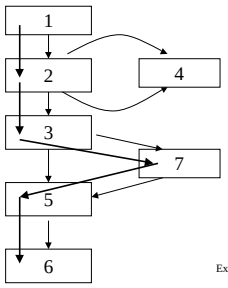
- Code Coverage needs basic block level instrumentation
  - dyninstAPI used to support function level instrumentation for sparc-solaris
  - added arbitrary instrumentation points for SPARC
- More state must be maintained in base trampolines
  - save/restore condition codes before/after arbitrary instrumentation points
  - Sparc arch supports user mode condition-code write/read for version v8plus and later

Using dominator tree

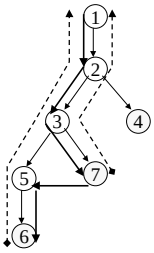
- Definitions
  - A **dom** B if all paths from entry to B goes through A
  - A **idom** B if (C != A) and (C dom B) implies (C dom A) for all C
  - dominator tree built using the **idom** relation
- Fact
  - execution of leaf node in dominator tree guarantees execution of all basic blocks along the path from root node

Instrumenting leaf nodes

CFG



Dominator Tree

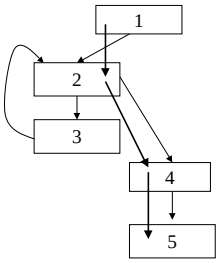


Executed:1, 2, 3, 5, 6, 7 Unexecuted: 4

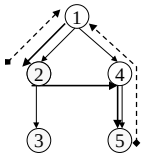
* instrument basic blocks that are leaf nodes in dominator tree

Instrumenting non-leaf node

CFG



Dominator Tree



Incorrect : Executed : 1, 4, 5 Unexecuted: 2, 3

Correct : Executed : 1, 2, 4, 5 Unexecuted: 3

* instrument basic blocks that have outgoing edge(s) to basic blocks not dominated by them

Code Coverage Algorithm

- Pre Run Phase
  - build CFG for each function
  - fill dominator info for CFG graph
  - choose basic blocks to instrument
  - assign a boolean variable to each basic block
  - at entry point to CFG set all variables to **false**
  - at the beginning of each basic block set its flag to **true**
  - use exit callback to record results

Code Coverage Algorithm(cont'd)

- Post Run Phase
  - at program termination for each basic block instrumented
    - read its variable
    - propagate the value up dominator tree
  - print the source code line numbers for the executed basic blocks
    - working on an improved UI

© 2000 tikir@cs.umd.edu

Dynamic Deletion

- Each Block Requires:
  - an instrumentation point with a base trampoline
  - each instrumentation point has a mini tramp
  - jump/call instructions
    - from mutatee address space to base tramp
    - from base tramp to mini tramp
- This is expensive for hot blocks!
- Once a block runs, don't need the code

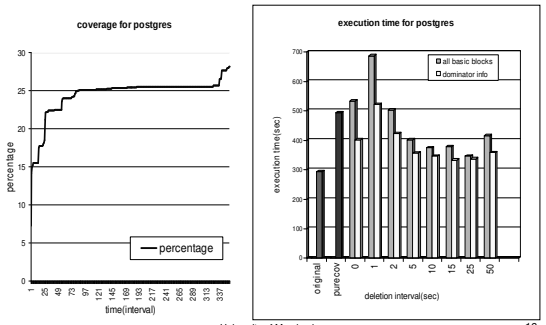
Instrumentation Code Deletion Policies

- Possible policies
  - at fixed time intervals given as parameter
  - first time it is executed
  - based on a cost-model
    - if inside a loop, how many times it will be executed, what the nest is, etc.
    - cost of deletion versus cost of execution
- We delete at fixed time intervals
  - stop the execution of mutatee
  - propagate information and delete instrumentation code for executed basic blocks

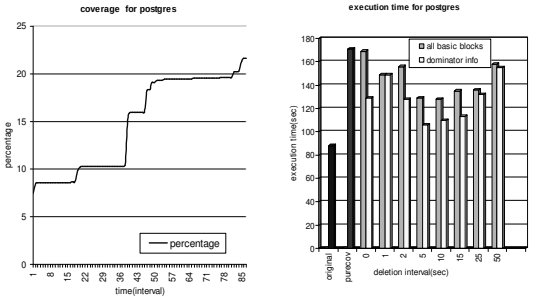
Current Status

- Fully implemented on sparc-solaris platform
- Works for executables compiled with gnu-C and native-C compilers
- Tested on
  - PostgreSQL object-relational DBMS
  - SPEC95/CINT spec benchmark in C
    - 099.go 124.m88ksim 126.gcc 129.compress
    - 130.li 132.ijpeg 134.perl 147.vortex

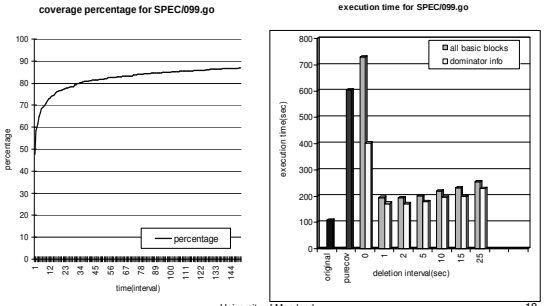
Postgres with crash-me benchmark



Postgres with wisconsin benchmark



SPEC95/099.go results



Total & Instrumented Basic Block Count

- Using dominator information reduces the total number of basic blocks instrumented by 35-50%

	Total number of Basic Blocks	Instrumentation count		Reduction Percentage %
		Leaf	Non-leaf	
Postgres	45028	22956	3350	46.1
099.go	11233	4571	1916	42.3
124.m88ksim	5708	2831	546	40.8
126.gcc	68448	28911	13866	37.5
129.compress	269	126	12	48.7
130.li	2532	1229	223	42.7
132.ijpeg	5670	2626	357	47.4
134.perl	13181	6695	1432	38.3
147.vortex	19047	8137	4442	34.0

University of Maryland 19

Conclusion

- Using dominator tree information
  - reduces number of inst. points by 35-50%
  - frequently outperforms **purecov**'s execution
    - **purecov** slows down execution up to 10 times
- Using all basic blocks also outperforms purecov's execution for some values of deletion interval
- Deletion of instrumentation code produces faster code coverage results

University of Maryland 20

Contribution

- Dynamic code insertion and deletion
  - existing code coverage tools use static code editing during/after compilation
  - instrumentation code is executed even though no extra information is produced
- Usage of dominator tree to reduce number of instrumentation
- Faster code coverage results for long running programs
- Less overhead for programs which have
  - Many infrequently executed paths
  - Many frequently executed paths

University of Maryland 21