Code Coverage Using DyninstAPI

- Identifying source code lines that are not executed

Motivation

- Needed for ensuring bug-free software
  - find lines of source code not executed
  - make sure that each path is taken at least once during the testing phase
  - detect semantic errors, e.g. part of the code is not supposed to be executed
- Useful for identifying bottlenecks in basic block level
  - use counters at the beginning of each basic block

Extension to dyninstAPI

- New classes for dyninstAPI
  - **BPatch_basicBlock** : machine code basic block
  - **BPatch_basicBlockLoop** : machine code natural loop
  - **BPatch_sourceBlock** : source code segment corresponding to machine basic block
  - **BPatch_flowGraph** : machine code control flow graph

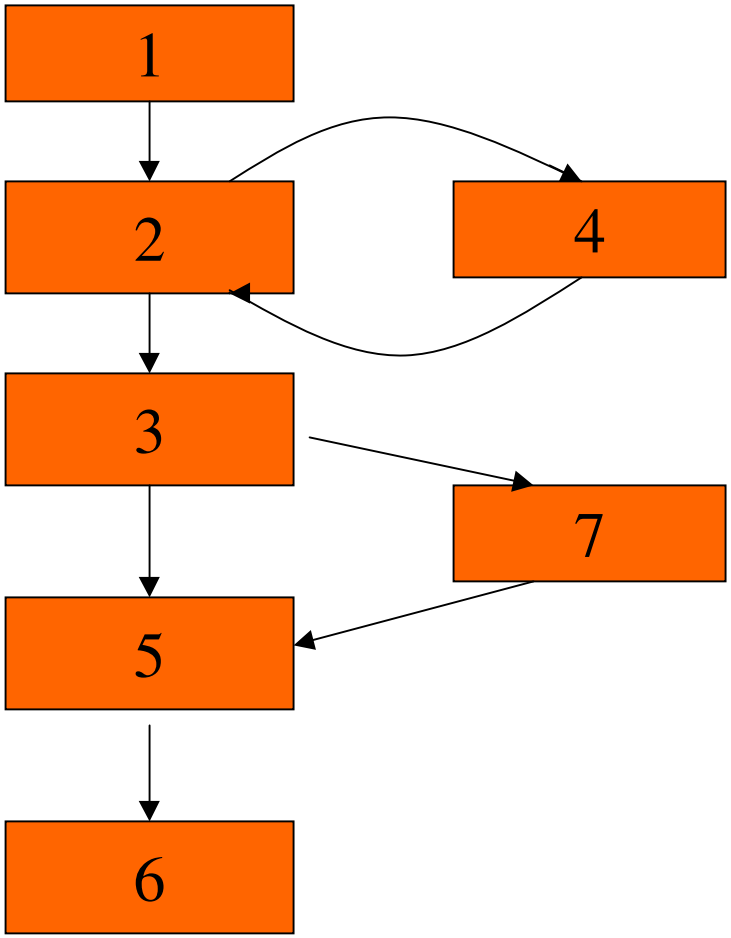
- Only creation of basic blocks are machine specific
  - machine specific functions for machine independence
  - classical optimizing compiler algorithms for dominators, loops

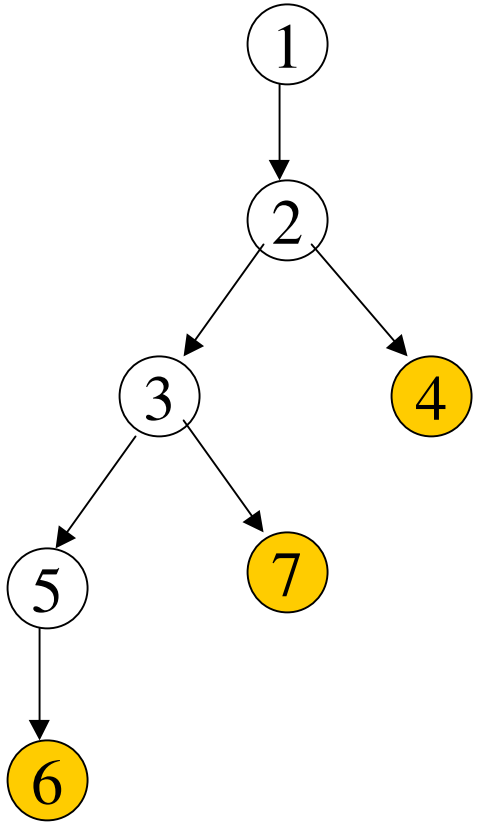
Extension to dyninstAPI(cont)

- Code coverage needs instrumentation points at basic block level
  - dyninstAPI supports entry, exit and call point instrumentation
  - added instrumentation points at arbitrary addresses
- Conservative base trampoline is needed
  - condition codes need not to be saved at entry, exit and call points of the functions
  - condition codes must be saved/restored before/after arbitrary instrumentation points

Pre run phase

- Build control flow graph , dominator tree and source blocks for machine basic blocks
- Choose basic blocks to be instrumented
  - basic blocks that are leaf nodes in dominator tree
    - if a basic block is executed, all dominators are also executed
    - less number of instrumentation
    - not applicable for counting execution times of basic blocks
- Create a flag for each basic block chosen


1
2
4
3
7
5
6


1
2
3
4
5
7
6

Pre run phase(cont)

- Instrument at function entry to set all flags to **false**
- Instrument at beginning of basic blocks to set corresponding flag to **true**
- Instrument breakpoints at **main** function exits, and **exit** system call entry

Post run phase

- At breakpoint, for all basic blocks instrumented
  - read the value of flag for the basic block
  - Propagate the value up the dominator tree
    - all basic blocks along the path from root to node are guaranteed to be executed
- Print the source code line numbers for the executed basic blocks

Can be done better?

- We have to delete instrumentation code for better performance
  - each instrumentation point has a base trampoline
  - each instrumentation code has a mini trampoline
  - there are jump instructions
    - from mutatee address space to base trampoline
    - from base trampoline to mini trampolines
    - jump instructions are expensive

Can be done better?(cont)

- Instrumentation code can be deleted
  - in fixed time intervals given as system parameter,
  - the first time it is executed and information is propagated,
  - based on a cost-model
    - if inside a loop, how many times it will be executed
    - the cost of the instrumentation code
    - the cost of deletion versus cost of execution, etc
- We delete at fixed time intervals
  - stop the execution of mutatee
  - delete instrumentation code for executed basic blocks

Contributions

- Dynamic code insertion and deletion
  - existing code coverage tools are based on static code editing during/after compilation
  - instrumentation code is never deleted
- Faster code coverage results for long running programs
  - less overhead for programs
    - which has a monotonically increasing coverage profile
    - which has many whole program paths and hot paths

Current Status

- Fully implemented on **sparc-sun-solaris-2.6** platform
- Mostly implemented on **mips-sgi-irix6-4** platform
- Testing code coverage software on long running programs
  - PostgreSQL object-relational database system
  - Apache Web Server