A New Scheme for Performance Consultant Searches in the Code Hierarchy

Matthew Cheyney
mcheyney@cs.wisc.edu

Computer Science Department
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706-1685
USA

Overview

- What is the Performance Consultant?
- The W^3 Search Model.
- Finding Performance Bottlenecks Automatically.
- Proposed Scheme.
- Technical Aspects.

What is the Performance Consultant?

- Motivation : finding performance bottlenecks in parallel applications is difficult.
- PC : system for automated performance evaluation and bottleneck detection in parallel systems.
- Uses W^3 Search Model, dynamic instrumentation, and iterative process of formulating, testing, and refining hypotheses to automatically find performance bottlenecks on the fly.

A Brief History of the PC

- Original Idea
  - Jeff Hollingsworth & Bart Miller
- First Implementation (demo'ed at SC'94)
  - Jeff Hollingsworth
- Major Re-engineering for Release 1.0 (scalable display, parallel search, etc.)
  - Karen Karavanic
- Next generation search (talked about here)
  - Matt Cheyney

The W^3 Search Model

- 3 Questions:
  - Why is the application performing poorly?
    - Formulate and test hypotheses (guesses about types of activity involved in the bottleneck)..
  - Where is the performance problem?
    - Isolate bottlenecks to specific resources (items - code or data used by the program for computation, communication, or IO) or resource hierarchies.
  - When does the problem occur?
    - Isolate bottlenecks to specific times or phases in program execution.

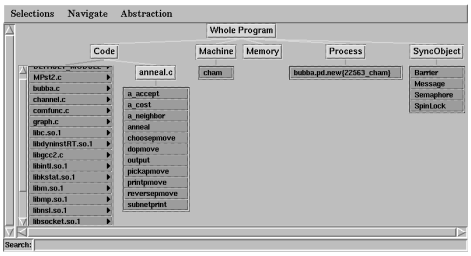
Why?

- Why Axis : Hierarchical representation of hypotheses - reasons for performance problems.
- Each why axis node (hypothesis) associated with specific performance metric and threshold value.
  - Test hypothesis: Performance Metric X > Threshold Y
- Examples:
  - CPU Bound - "cpu" metric : fraction of time spent computing during time interval.
  - Sync Bound - "sync_wait" metric : fraction of time spent waiting for synchronization during time interval.
  - IO Bound - "io_wait" metric : fraction of time spent blocking on IO during time interval.

© 1998 mcheyney@cs.wisc.edu

Where?

- Where Axis: Hierarchical representation of foci - classes of resources.



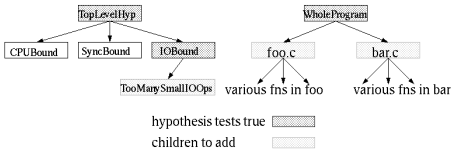
Basic Search Process

- Designed to match process programmers actually use.
  - Formulate and test top level hypothesis for whole program.
  - If true, refine search to specific parts of program.
  - Top down, efficient.
- Theory why program is slow.
  - Involves hypothesis (reason) from why axis, focus (resource hierarchy) from where axis, and phase (time) from when axis.
  - Initial theory:
    - WholeProgram is CPUBound | SyncBound | IOBound.

The Basic Process (continued)

- Test hypothesis, focus (reason, resource) pair.
  - Instrument program to collect data for pair.
    - e.g. use "io_wait" metric on module "libc".
  - Collect data.
- True
  - Refine search.
- False
  - Discard guess and try another.
- Refinement
  - (h, f) pair tests true : add all child hypotheses of h with f as focus, and add all child resources of f with h as hypothesis.

Refinement Example



- Example: WholeProgram is bound by IOBlocking:
  - foo.c is bound by IOBlocking.
  - bar.c is bound by IOBlocking.
  - WholeProgram is bound by TooManySmallIOOps.

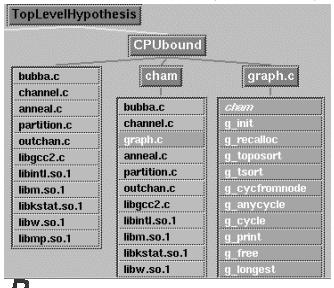
Refining Code Hierarchy (current)

- Narrow "where" question by refining on where axis.
  - Where axis includes Code Hierarchy.
  - Code Hierarchy refines code into modules, refines modules into functions.
- Refining on code hierarchy didn't work as expected:

Refining Code Hierarchy (current)

- Too many modules (tree too flat).
  - Original idea : few modules, many functions.
  - Actual organization : many modules, many functions.
- Module instrumentation not cheaper than function instrumentation.
  - To instrument module, instrument all functions.
- Not related to program execution pattern.
  - Doesn't provide mechanism of focusing search - tries all metric/focus pairs (including uncalled functions).
- Uses expensive metrics.
  - Inclusive vs. Exclusive metrics.

Refining Code Hierarchy (current)

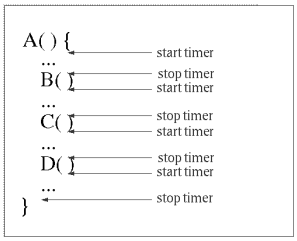


- anneal.c: 11 fns
- bubba.c: 1 fn
- channel.c: 13 fns
- graph.c: 12 fns
- outchan.c: 5 fns
- partition.c: 11 fns
- total user code : ~ 50 fns
- libc: >1000 fns
- libw: ~175 fns
- libsocket: ~150fns
- libintl: ~35 fns
- libm: ~150 fns
- libkstat: ~10 fns
- libmp: ~40 fns
- libnsl: ~1300 fns
- total library code : ~3000 fns

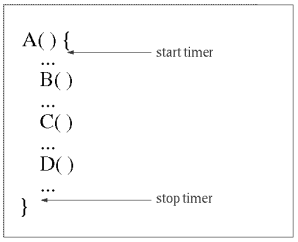
Different Metrics

- Inclusive vs. Exclusive Metrics
  - Exclusive : Include all time spent in function, but not in functions called from it.
    - Add instrumentation to beginning, end of function, and function call sites and returns from call sites.
  - Inclusive : Include all time spent in function and functions called from it.
    - Add instrumentation to beginning, end of function.

Instrumenting Exclusive Metrics



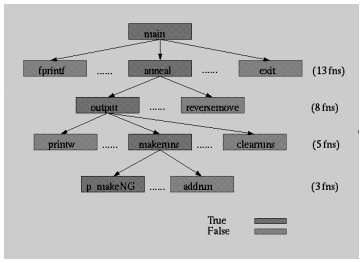
Instrumenting Inclusive Metrics



Refining Code H. (proposed)

- Use call graph to pick next place in code hierarchy to search.
  - When a hypothesis is true for a given function, refine to functions called by it.
- Advantages:
  - Refinement based on actual program structure.
    - Avoid flat, static module and function structure.
    - Call tree nodes tend to have fewer children.
  - Can use cheaper metrics.
    - Inclusive vs. Exclusive Metrics
  - Avoid unused functions.
    - Tries only functions which are used in code.

Refining Code H. (proposed)



Technical Challenges

- Use static analysis and dynamic instrumentation to build call graph.
  - Build as much as possible with static analysis.
  - Use dynamic instrumentation for the rest.
    - Jumps to computed addresses, jumps to loaded addresses, pointers to functions, shared libraries.
    - Using dynamic instrumentation adds overhead (once).
  - Can build call graph lazily - only include functions when they are called.
    - Function is never XBound before it is called.

Conclusion

- New scheme for code hierarchy refinement better suited to Performance Consultant because:
  - Current hierarchy too flat.
  - Can be implemented with cheaper metrics.
  - Smarter targeted search, uses hints from program execution/structure.
- Demo version 2Q'98.

When?

- When Axis : Division of program into phases.
  - Why are they necessary?
    - Programs may exhibit different behavior at different times in execution, thus have different bottlenecks.
  - How To Find?
    - User specified (current)
    - Based on predicates and state detection.
- Simple example :
  - (1) Initialization (I/O or synchronization bound).
  - (2) Work: repeated
    - (2a) Computation (CPU bound).
    - (2b) Communication (synchronization bound).