SAPIENZA
UNIVERSITÀ DI ROMA

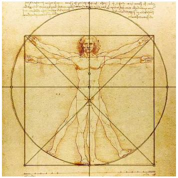
Software Streams

Big Data Challenges in Dynamic Program Analysis

Irene Finocchi
Dept. Computer Science – Sapienza U. Rome

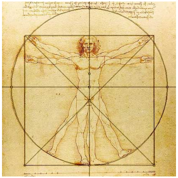
Theory versus practice

Theory is when you know something, but it doesn't work.



Theory versus practice

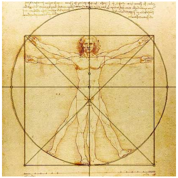
Theory is when you know something, but it doesn't work.



Practice is when something works, but you don't know why.

Theory versus practice

Theory is when you know something, but it doesn't work.



Practice is when something works, but you don't know why.

Programmers combine theory and practice: Nothing works, and they don't know why.

(Anonymous)

Topic of the talk

Algorithm engineering talk: boosting practice with theory

Topic of the talk

Algorithm engineering talk: boosting practice with theory

Theory: data stream algorithmics

Application area: dynamic program analysis

Program analysis

Development of techniques and tools for analyzing the structure and the behavior of a software system

Program analysis

Development of techniques and tools for analyzing the structure and the behavior of a software system

Goals:

- conclude properties about the program: e.g., correctness, resource consumption
- seek opportunities for optimization
- error detection and correction: e.g., type checking, memory safety, data structure repair, protection against security attacks
- study how the program or its parts are used: e.g., usage patterns, intrusion detection
- program understanding

Static vs. dynamic analysis

Static analysis: based on knowledge of code (source, object, ...)

Examples:

- compilers
- formal verification systems
- theoretical analysis of algorithms

Static vs. dynamic analysis

Static analysis: based on knowledge of code (source, object, ...)

Examples:

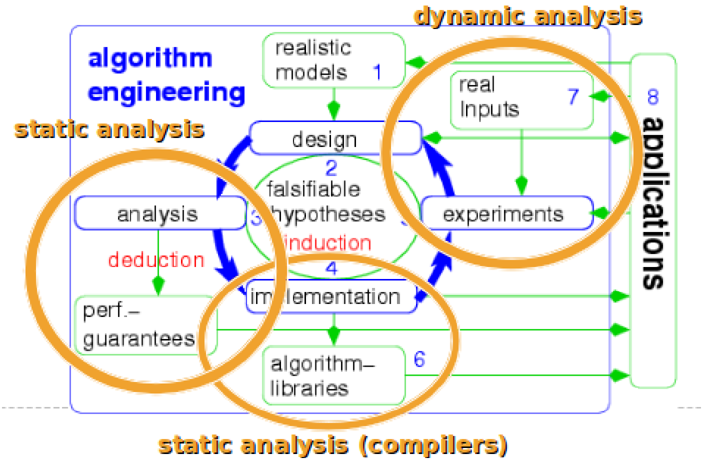
- compilers
- formal verification systems
- theoretical analysis of algorithms

Dynamic analysis: exploits information gathered at runtime

Examples:

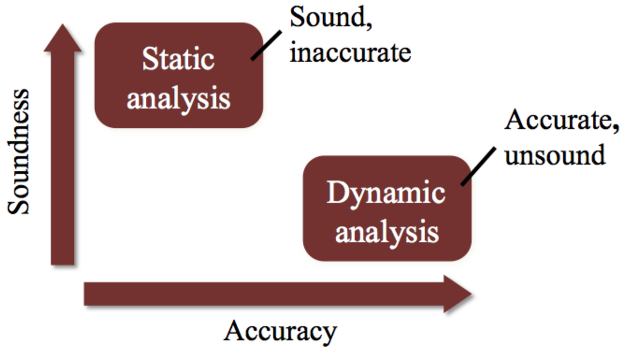
- debuggers, memory checkers
- performance profilers
- platforms for the experimental evaluation of algorithms

Program analysis in algorithm engineering



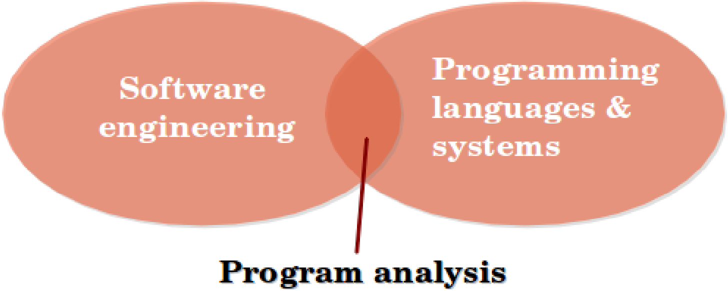
Soundness vs. accuracy

Static analysis huge success in software design,
but dynamic nature of modern computing scenarios
makes it increasingly more inaccurate

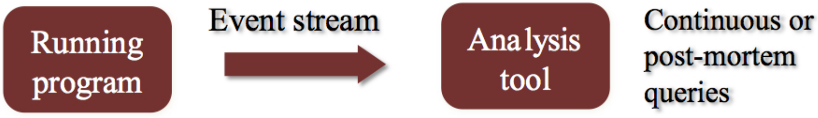


Program analysis community

Many disciplines involved:
programming languages, SE, architectures, algorithms, statistics. . .



This talk: algorithmics for dynamic program analysis



Events of interest:

- routine calls
- memory accesses
- low-level instructions
- ...
- system calls
- cache misses
- interrupts

What's difficult?

Capturing events

- hardware support (counters, watchpoints)
- programmable interrupts/signals
- program instrumentation (source code or binary code)

What's difficult?

Capturing events

- hardware support (counters, watchpoints)
- programmable interrupts/signals
- program instrumentation (source code or binary code)

Intrusiveness: Heisenberg effects (the act of observing a system causes the system to change)

What's difficult?

Capturing events

- hardware support (counters, watchpoints)
- programmable interrupts/signals
- program instrumentation (source code or binary code)

Intrusiveness: Heisenberg effects (the act of observing a system causes the system to change)

Performance: analysis inlined with program execution, slow down analyzed programs, real-time performance (billions of events per second)

What's difficult?

Capturing events

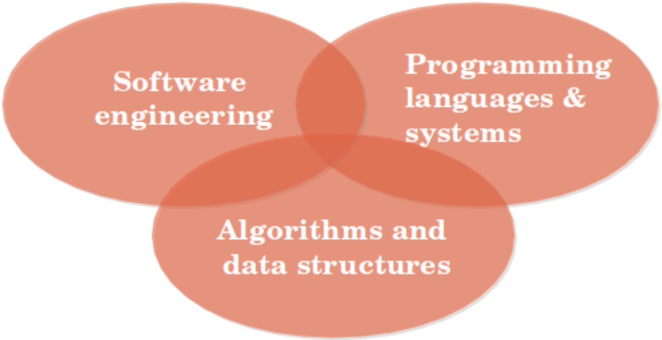
- hardware support (counters, watchpoints)
- programmable interrupts/signals
- program instrumentation (source code or binary code)

Intrusiveness: Heisenberg effects (the act of observing a system causes the system to change)

Performance: analysis inlined with program execution, slow down analyzed programs, real-time performance (billions of events per second)

Massive data: dynamic analysis tools process huge amounts of data, cannot store all of them

Efficient algorithms can make a difference



Automated dynamic analysis less explored than static analysis from an algorithmic perspective...

Software streams

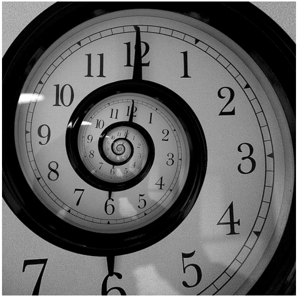
An example: performance profiling

Form of dynamic program analysis that typically measures:

- execution time of instructions, basic blocks, routines
- frequency of portions of code

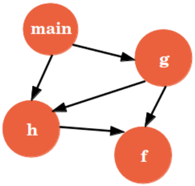
Our goal: identify routines that contribute most to the running time (hot routines)

Mainly useful for performance optimization



Profiler characteristics

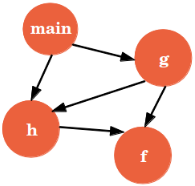
- Granularity
  - Basic blocks
  - Routines



Call graph

Profiler characteristics

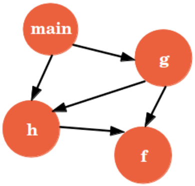
- Granularity
  - Basic blocks
  - Routines
- Metrics
  - Time
  - Number of routine calls
  - Cache misses, I/Os . . .



Call graph

Profiler characteristics

- Granularity
  - Basic blocks
  - Routines
- Metrics
  - Time
  - Number of routine calls
  - Cache misses, I/Os . . .



Call graph

- Data aggregation level
  - Vertex: how many times is routine f called?
  - Edge: how many times is f called from g?
  - Calling context: how many times is f called along path $main \to g \to h \to f$?

Vertex vs. calling context profiling

Vertex profiling:

- Stream $\Sigma =$
  $= \langle main, g, h, f, \ldots \rangle =$
  $= \langle f_1, f_2, \ldots, f_n \rangle$
- Item universe:
  $f_i \in \{routines\}$
- Query: find most frequently called routines

Vertex vs. calling context profiling

Vertex profiling:

- Stream $\Sigma =$
  $= \langle main, g, h, f, \ldots \rangle =$
  $= \langle f_1, f_2, \ldots, f_n \rangle$
- Item universe:
  $f_i \in \{routines\}$
- Query: find most frequently called routines

Calling context profiling:

- Stream $\Sigma = \langle main, main \rightarrow h, main \rightarrow g \rightarrow h \ldots \rangle =$
  $= \langle \pi_1, \pi_2, \ldots, \pi_n \rangle$
- Item universe:
  $\pi_i \in \bigcup_j^{\infty} \{routines\}^j$
- Query: find most frequent calling contexts

Conventional approaches

Keep complete profiling info about vertices or paths

Vertex profiling:

- Hash table
- Space required = Θ(number of distinct routines in Σ)

Conventional approaches

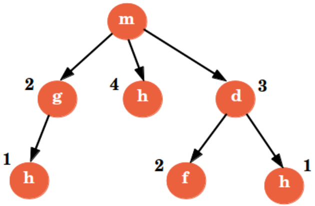
Keep complete profiling info about vertices or paths

Vertex profiling:

- Hash table
- Space required = Θ(number of distinct routines in Σ)

Calling context profiling:

- Calling context tree
- Space required = Θ(number of CCT nodes) = = Θ(number of distinct call paths in Σ)

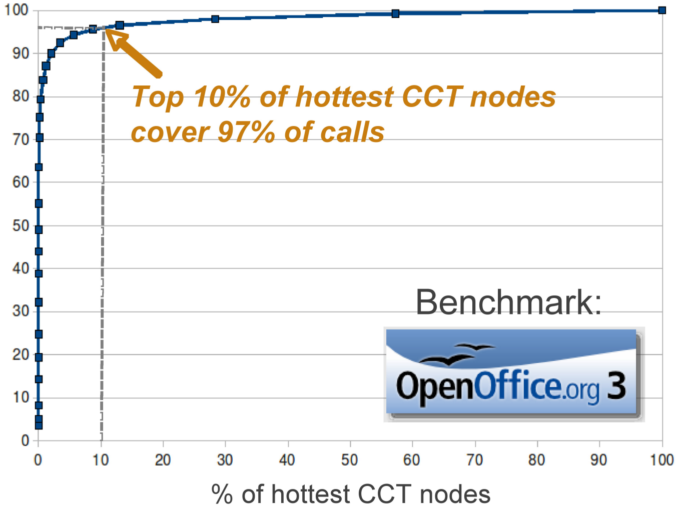


How much space?

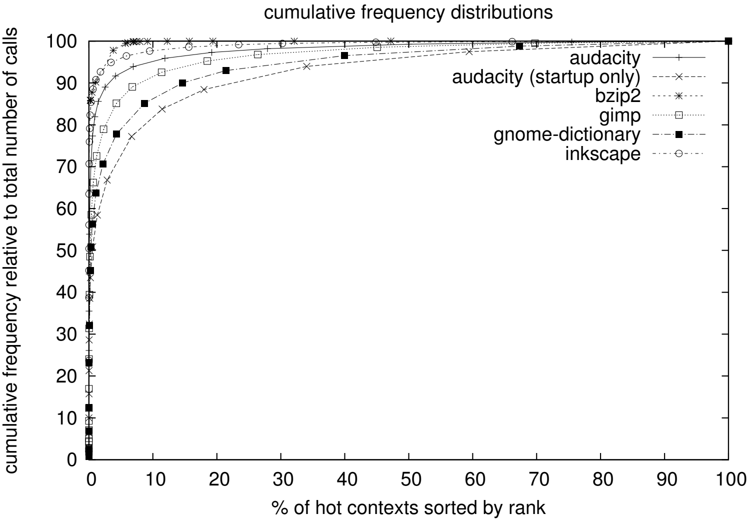
Application	|Call graph|	|Call sites|	|CCT|	|Σ|
amarok	13 754	113 362	13 794 470	991 112 563
audacity	6 895	79 656	13 131 115	924 534 168
bluefish	5 211	64 239	7 274 132	248 162 281
dolphin	10 744	84 152	11 667 974	390 134 028
firefox	6 756	145 883	30 294 063	625 133 218
gedit	5 063	57 774	4 183 946	407 906 721
gimp	5 146	93 372	26 107 261	805 947 134
sudoku	5 340	49 885	2 794 177	325 944 813
inkscape	6 454	89 590	13 896 175	675 915 815
oocalc	30 807	394 913	48 310 585	551 472 065
pidgin	7 195	80 028	10 743 073	404 787 763
quanta	13 263	113 850	27 426 654	602 409 403

- Runs of a few minutes of real applications produce Gigabytes of information
- Storing the CCT requires hundreds of Megabytes

Skewness


100
90
80
70
60
50
40
30
20
10
0
0 10 20 30 40 50 60 70 80 90 100
Top 10% of hottest CCT nodes
cover 97% of calls
Benchmark:
OpenOffice.org 3
% of hottest CCT nodes

Pareto principle (80-20 rule)



Patterns

Execution traces typically contain:

- several event repetitions, either contiguous or not
- a very large number of patterns
- each pattern can have thousands of occurrences

Data mining, pattern detection, and compression techniques very useful to understand the characteristics of execution traces

Case studies

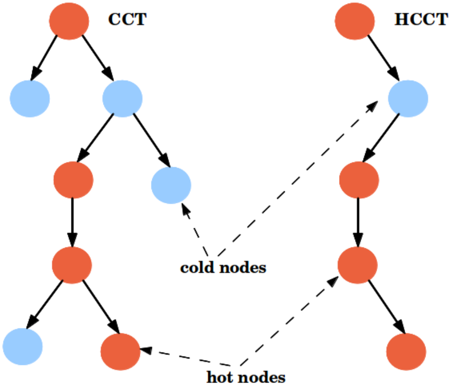
Mining hot calling contexts space-efficiently

Keep information about hot contexts only

Ignore on the fly info about contexts with low frequency

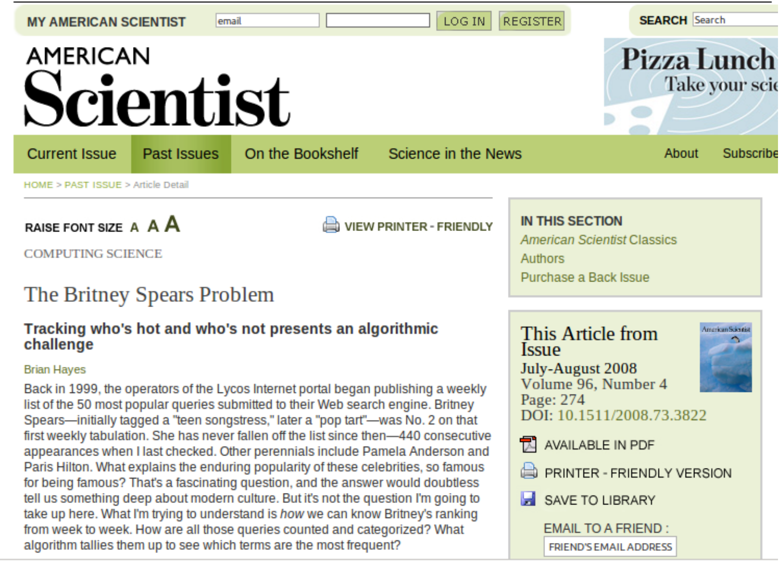
[D'Elia, Demetrescu & F., PLDI 2011]

Hot calling context tree



- The CCT unfolds during program execution
- How do we prune it on-line (to get the HCCT)?

The Britney Spears problem...



MY AMERICAN SCIENTIST email LOG IN REGISTER SEARCH Search

AMERICAN Scientist

Pizza Lunch
Take your scie

Current Issue | Past Issues | On the Bookshelf | Science in the News | About | Subscribe

HOME > PAST ISSUE > Article Detail

RAISE FONT SIZE A A A VIEW PRINTER - FRIENDLY

COMPUTING SCIENCE

The Britney Spears Problem

Tracking who's hot and who's not presents an algorithmic challenge

Brian Hayes

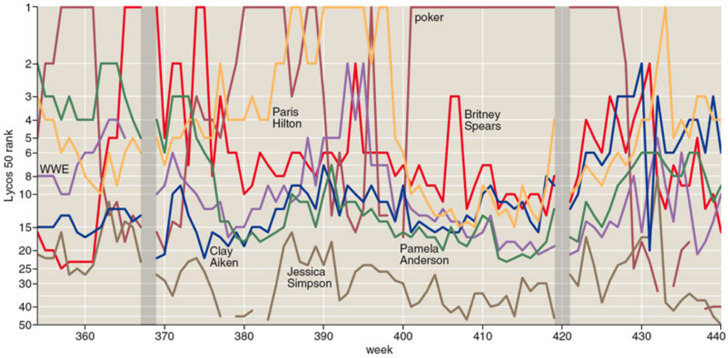
Back in 1999, the operators of the Lycos Internet portal began publishing a weekly list of the 50 most popular queries submitted to their Web search engine. Britney Spears—initially tagged a "teen songstress," later a "pop tart"—was No. 2 on that first weekly tabulation. She has never fallen off the list since then—440 consecutive appearances when I last checked. Other perennials include Pamela Anderson and Paris Hilton. What explains the enduring popularity of these celebrities, so famous for being famous? That's a fascinating question, and the answer would doubtless tell us something deep about modern culture. But it's not the question I'm going to take up here. What I'm trying to understand is *how* we can know Britney's ranking from week to week. How are all those queries counted and categorized? What algorithm tallies them up to see which terms are the most frequent?

IN THIS SECTION
American Scientist Classics
Authors
Purchase a Back Issue

This Article from Issue
July-August 2008
Volume 96, Number 4
Page: 274
DOI: 10.1511/2008.73.3822

AVAILABLE IN PDF
PRINTER - FRIENDLY VERSION
SAVE TO LIBRARY
EMAIL TO A FRIEND :
FRIEND'S EMAIL ADDRESS

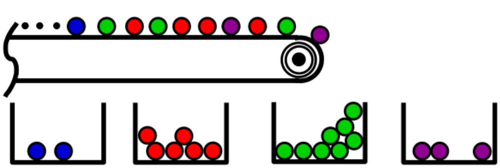
... tracking who's hot and who's not



"... can't just pay attention to a few popular subjects, because you can't know in advance which ones are going to rank near the top. To be certain of catching every new trend as it unfolds, you have to monitor *all* the incoming queries – and their variety is unbounded. "

Heavy hitters

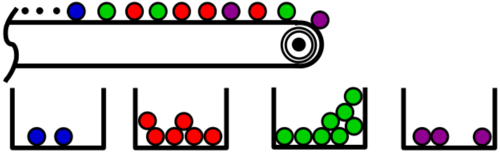
Given a stream of n items, find those that appear "most frequently"



E.g., items occurring more than 1% of the time

Heavy hitters

Given a stream of n items, find those that appear "most frequently"

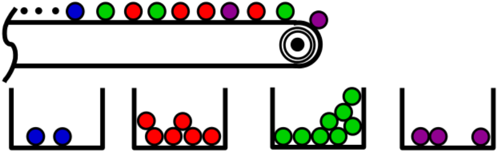


E.g., items occurring more than 1% of the time

- Formally "hard" in small space, so allow approximation

Heavy hitters

Given a stream of n items, find those that appear "most frequently"

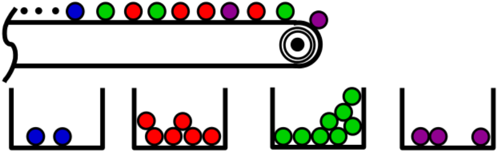


E.g., items occurring more than 1% of the time

- Formally "hard" in small space, so allow approximation
- No false negatives: return all items with count $\geq \varphi n$

Heavy hitters

Given a stream of n items, find those that appear "most frequently"

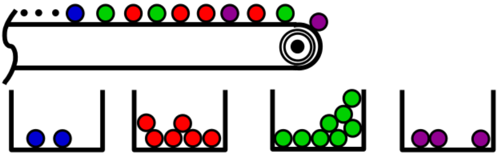


E.g., items occurring more than 1% of the time

- Formally "hard" in small space, so allow approximation
- No false negatives: return all items with count $\geq \varphi n$
- "Good" false positives: no item with count $< (\varphi - \varepsilon)n$ is returned (error $\varepsilon \in (0,1)$, $\varepsilon \ll \varphi$)

Heavy hitters

Given a stream of n items, find those that appear "most frequently"



E.g., items occurring more than 1% of the time

- Formally "hard" in small space, so allow approximation
- No false negatives: return all items with count $\geq \varphi n$
- "Good" false positives: no item with count $< (\varphi - \varepsilon)n$ is returned (error $\varepsilon \in (0, 1)$, $\varepsilon \ll \varphi$)
- Related problem: estimate each frequency with error $\pm\varepsilon n$

A well-studied problem

- Core streaming problem: connections with entropy estimation, itemsets mining, compressed sensing
- Extensive research: scores of streaming papers on frequent items and its variations

A well-studied problem

- Core streaming problem: connections with entropy estimation, itemsets mining, compressed sensing
- Extensive research: scores of streaming papers on frequent items and its variations

Two approaches:

1. Sketch-based
   - Maintain a sketch of the whole data set
   - Estimate frequency of both frequent and non-frequent items

A well-studied problem

- Core streaming problem: connections with entropy estimation, itemsets mining, compressed sensing
- Extensive research: scores of streaming papers on frequent items and its variations

Two approaches:

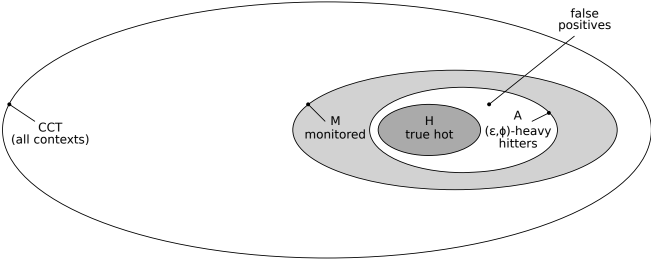
1. Sketch-based
   - Maintain a sketch of the whole data set
   - Estimate frequency of both frequent and non-frequent items
2. Counter-based
   - Maintain estimated counters of frequent items only
   - Work very well on skewed input distributions

(Some) counter-based algorithms

1. Sticky sampling Gibbons & Matias, SIGMOD 1998 – Manku & Motwani, VLDB 2002]
    - probabilistic, sampling-based approach
    - correct with probability $\geq 1 - \delta$, with $\delta \in (0, 1)$ user-specified probability of failure
    - space $O(\frac{1}{\varepsilon} \cdot \log \frac{1}{\varphi\delta})$
2. Lossy counting [Manku & Motwani, VLDB 2002]
    - deterministic
    - space $O(\frac{1}{\varepsilon} \cdot \log(\varepsilon n))$
3. Space saving [Metwally, Agrawal & El Abbadi, ACM TODS 2006]
    - deterministic
    - space $O(\frac{1}{\varepsilon})$ (provably optimal)

Back to hot calling contexts

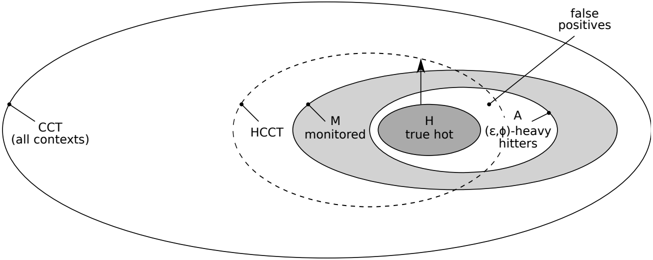
- Maintain a set M of monitored calling contexts
- Upon query, return a subset $A \subseteq M$: $A = \{\ (\varphi, \varepsilon)\text{-heavy hitters}\}$
- All true hot contexts are returned: $H \subseteq A$ (no false negatives)
- False positives are "good"



- The CCT induces a tree structure over sets H, A, M

Back to hot calling contexts

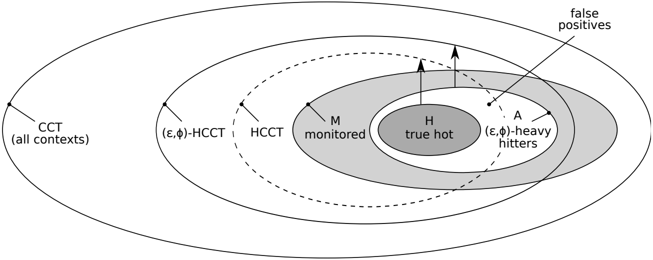
- Maintain a set M of monitored calling contexts
- Upon query, return a subset $A \subseteq M$: $A = \{$ (φ, ε)-heavy hitters$\}$
- All true hot contexts are returned: $H \subseteq A$ (no false negatives)
- False positives are "good"



- The CCT induces a tree structure over sets H, A, M

Back to hot calling contexts

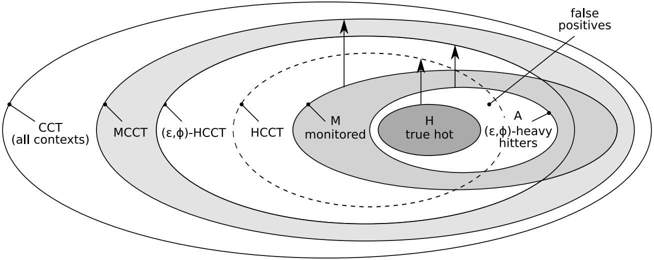
- Maintain a set M of monitored calling contexts
- Upon query, return a subset $A \subseteq M$: $A = \{$ (φ, ε)-heavy hitters$\}$
- All true hot contexts are returned: $H \subseteq A$ (no false negatives)
- False positives are "good"



- The CCT induces a tree structure over sets H, A, M

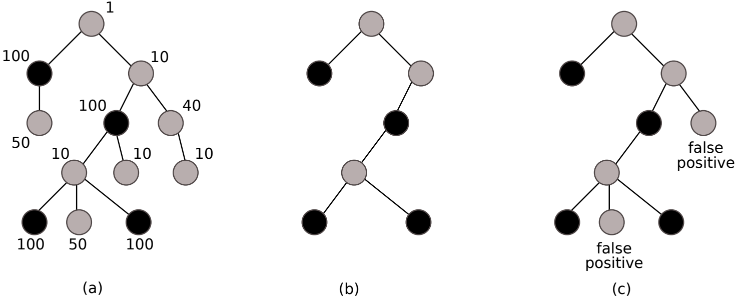
Back to hot calling contexts

- Maintain a set M of monitored calling contexts
- Upon query, return a subset $A \subseteq M$: $A = \{ (\varphi, \varepsilon)$-heavy hitters$\}$
- All true hot contexts are returned: $H \subseteq A$ (no false negatives)
- False positives are "good"



- The CCT induces a tree structure over sets H, A, M

(φ, ε)-hot calling context tree



(a) CCT: entire calling context tree

(b) HCCT: hot calling context tree

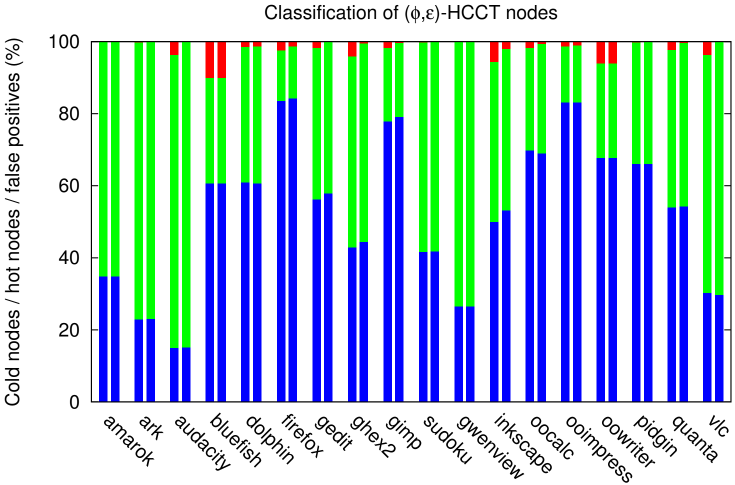
- hot nodes
- cold internal nodes

(c) (φ, ε)-HCCT: (φ, ε)-hot calling context tree

- hot nodes
- cold internal nodes
- "almost hot" leaves (false positives)

How many false positives?

Lossy Counting (left bars) versus Space Saving (right bars)



Parameter tuning

Rule of thumb: $\varepsilon = \varphi/10$ [Cormode and Hadjieleftheriou, PVLDB 2008]

Parameter tuning

Rule of thumb: $\varepsilon = \varphi/10$ [Cormode and Hadjieleftheriou, PVLDB 2008]

$\varepsilon = \varphi/5$ works great here due to distribution skewness

Parameter tuning

Rule of thumb: $\varepsilon = \varphi/10$ [Cormode and Hadjieleftheriou, PVLDB 2008]

$\varepsilon = \varphi/5$ works great here due to distribution skewness

What about φ?

Parameter tuning

Rule of thumb: $\varepsilon = \varphi/10$ [Cormode and Hadjieleftheriou, PVLDB 2008]

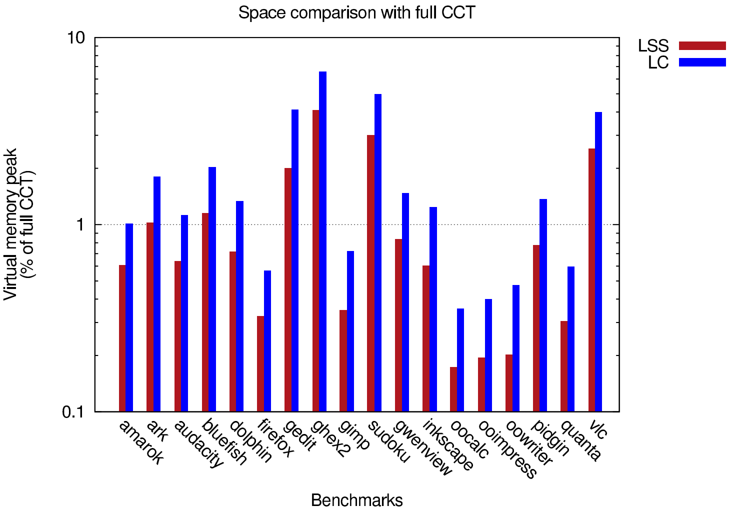
$\varepsilon = \varphi/5$ works great here due to distribution skewness

What about φ?

Benchmark	HCCT nodes $\phi = 10^{-3}$	HCCT nodes $\phi = 10^{-5}$	HCCT nodes $\phi = 10^{-7}$
audacity	112	9 181	233 362
dolphin	97	14 563	978 544
gimp	96	15 330	963 708
inkscape	80	16 713	830 191
oocalc	136	13 414	1 339 752
quanta	94	13 881	812 098

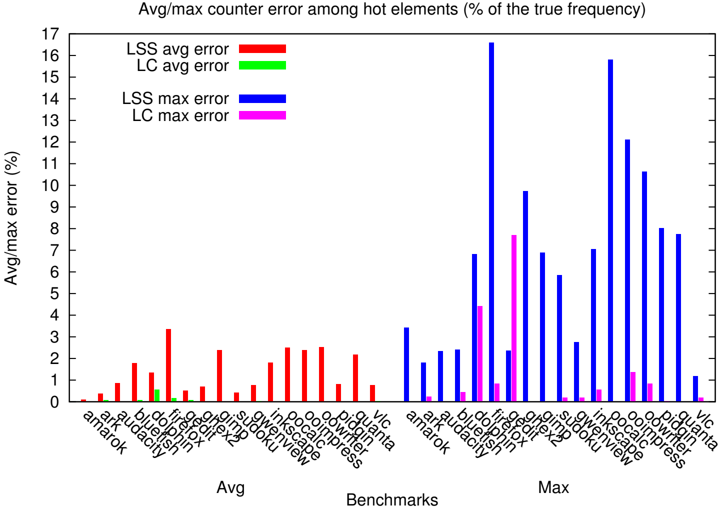
Space analysis

Space Saving (LSS) vs. Lossy Counting (LC): $\varphi = 10^{-4}$, $\varepsilon = \varphi/5$



Counter accuracy

Space Saving (LSS) vs. Lossy Counting (LC): $\varphi = 10^{-4}$, $\varepsilon = \varphi/5$



Other applications

Range adaptive profiling

Assume, e.g., that we want to profile lines of code: if 90% of time is spent on the top half of the code, fine-grained profile data on the bottom half would not be very useful

Range adaptive profiling

Assume, e.g., that we want to profile lines of code: if 90% of time is spent on the top half of the code, fine-grained profile data on the bottom half would not be very useful

Output profile data into a hierarchical fashion, grouping data into ranges: [Mysore *et al.*, CGO 2006]

- most frequent ranges broken down into subranges
- least frequent events kept as larger ranges

Range adaptive profiling

Assume, e.g., that we want to profile lines of code: if 90% of time is spent on the top half of the code, fine-grained profile data on the bottom half would not be very useful

Output profile data into a hierarchical fashion, grouping data into ranges: [Mysore *et al.*, CGO 2006]

- most frequent ranges broken down into subranges
- least frequent events kept as larger ranges

Adaptive Spatial Partitioning (ranges and their counters stored in a tree): [Hershberger *et al.*, Algorithmica 2006]

- when range gets sufficiently hot, corresponding tree node split into subranges
- ranges that get colder are merged together, pruning the tree

Uniform sampling over software streams

- Sampling very popular to reduce size of execution traces and runtime overhead of dynamic analysis tools

Uniform sampling over software streams

- Sampling very popular to reduce size of execution traces and runtime overhead of dynamic analysis tools
- Main approach in state-of-the-art profilers: fixed rate sampling (e.g., take one item every 10 ms)
  - Pros: easy to implement
  - Cons: produce biased samples when the original trace exhibits regular patterns

Uniform sampling over software streams

- Sampling very popular to reduce size of execution traces and runtime overhead of dynamic analysis tools
- Main approach in state-of-the-art profilers: fixed rate sampling (e.g., take one item every 10 ms)
  - Pros: easy to implement
  - Cons: produce biased samples when the original trace exhibits regular patterns
- To avoid bias and get representative samples, need uniform sampling probability [Mytkowicz *et al.*, PLDI 2010]

Uniform sampling over software streams

- Sampling very popular to reduce size of execution traces and runtime overhead of dynamic analysis tools
- Main approach in state-of-the-art profilers: fixed rate sampling (e.g., take one item every 10 ms)
  - Pros: easy to implement
  - Cons: produce biased samples when the original trace exhibits regular patterns
- To avoid bias and get representative samples, need uniform sampling probability [Mytkowicz *et al.*, PLDI 2010]
- Randomized approaches (e.g., reservoir sampling [Vitter, ACM TMS 1985]) leverage these issues [Coppa *et al.*, 2013]

Conclusions

Dynamic program analysis:

- data-intensive nature makes it great source of algorithmic problems
- a lot of fun with algorithms, systems, and architectures
- automated analysis provides valuable tools in algorithm engineering

Challenges: analysis of programs on multi-core platforms, big data applications, and resource-constrained systems