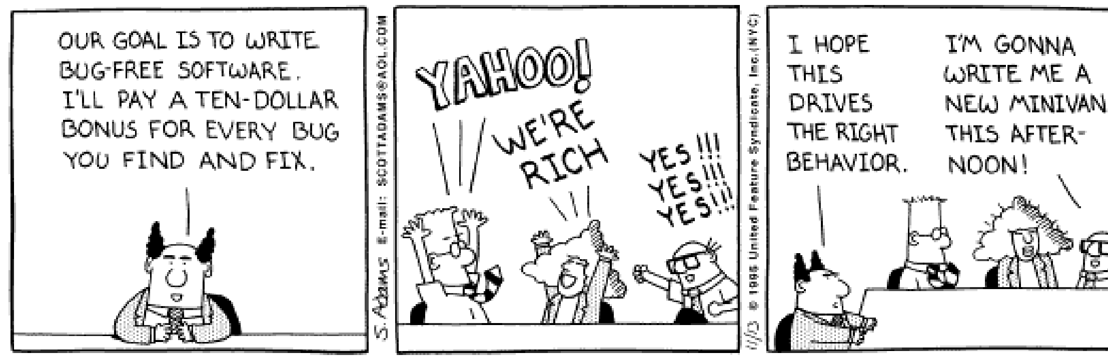
Quality

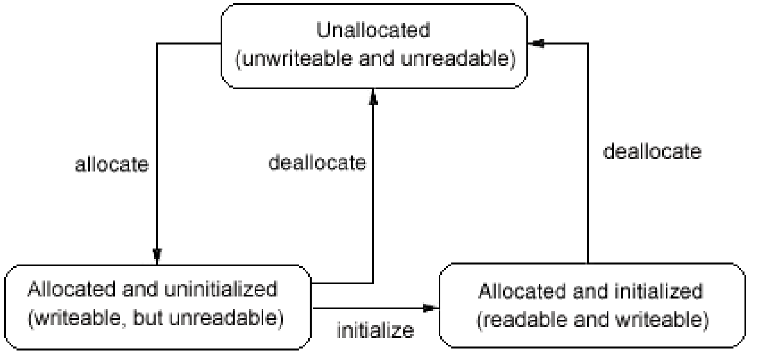


Avoiding memory problems - memprof

Avoiding performance problems - perfanal

The larger picture

Memory-related problems



Read/write incorrectly

- Read from uninitialized memory
- Read via uninitialized pointer/reference
- Read/write past the valid range
- Read/write via a stale pointer/reference
  - E.g. after deallocating memory

Memory management mistakes

- Deallocation of (currently) unowned memory
  - Freeing something twice results in later overwrites
- Memory leaks
  - Forgetting to free something results in unusable memory

Often cause "really hard to find" bugs

- Crashes, incorrect results - traceback, dump don't show cause
- Occur far from the real cause - breakpoints don't help
- Often intermittent

Note: Java reduces these, but doesn't make them go away!

A better allocator (malloc) can find some of these

Standard GNU malloc has a run-time checking option:

```
$ a.out
Segmentation fault (core dumped)
$ setenv MALLOC_CHECK_ yes
$ a.out
malloc: using debugging hooks
free(): invalid pointer 0x8049840!
```

Why not always leave it set?

- Checking slows program significantly
- Too many errors?

3rd party tools exist to do an even better job

```
Finished  a.out                ( 583 errors,  182 leaked bytes)
  Purify/PureCoverage instrumented a.out (pid 9499 at Mon Sep 23 14:07:32 1996)
  UMR: Uninitialized memory read (13 times)
    This is occurring while in:
        putHash        [hash.c:134]
              char* new_key;
              void* old_value;
              int index = hashIndex(key);
          ⇨   for (last_entry = NULL, entry = ht[index];
                   entry && strcmp(entry->key, key);
                   last_entry = entry, entry = entry->next) {
              }
        testPutHash    [testHash.c:84]
```

Specialized tools - leak checking

Automated, unambiguous identification of leaks is difficult

- “forgot to free” vs “haven’t freed yet” vs “program’s ending, don’t bother”
- “can no longer reference any part” vs “no references to the beginning”

But reading the code is not a reliable method either

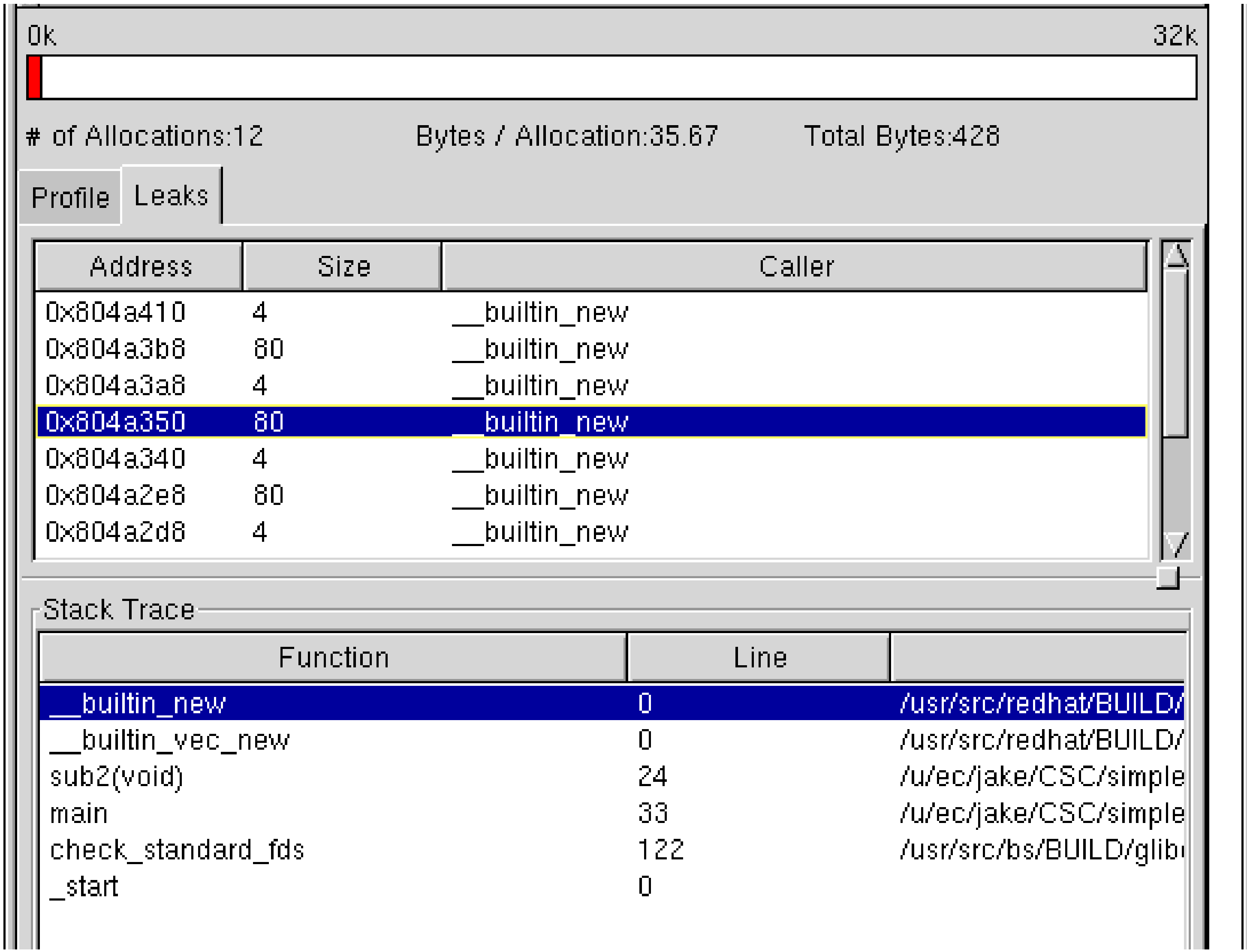
- A leak is a mistake of omission, not commission
- Often requires cooperation to leak memory:
  - Creator of allocated item may have no idea where it goes
  - Consumer may not realize responsible for deallocation
    - Doesn’t need to be deallocated
    - Expects some third party to deallocate

Several approaches:

- “Print it all, and let the human sort it out”
- Provide a browser, let human reason about status of remaining memory
- Provide a suite of heuristics that can be tuned to the code’s structure

Example: memprof

memprof replaces the allocation library at runtime, provides simple GUI



How do these actually work?

Replacement libraries

- E.g. a more careful malloc, perhaps automatically linked
- Can’t check individual load/store instructions

Source code manipulation

- Preprocessor inserts instrumentation before compilation

  Can know about scope, variable accesses, control flow

  But requires source code, is language specific

Object code insertion

- Process object code to recognize & instrument load/store instructions

  Can efficiently check every use of memory

  Specific to both architecture and compiler, hard to port

Yes, you can write your own code to do some of this

But do you really want to spend the time to do it well?

A small catalog of available memory tools

Free validity tests

- GNU C library - enable checking via MALLOC_CHECK_
- DMalloc - replacement library with instrumentation
- ElectricFence - checks for write outside proper boundaries
- valgrind - instruction-by-instruction checking

Free leak checkers

- Boehm GC
- Debauch
- Memprof
- LeakTracer
- ccmalloc

Commercial code-check suites

- Purify (Rational Software)
- Insure (Parasoft)

How do you use these?

Big-bang approach is incredibly depressing

- Familiar products have thousands of memory errors
- These swamp your own tiny efforts

Better: isolate your own code for initial checks

- Ties in with a test framework: “Does it work as expected?”

You still have to test “in the wild”

- Many errors are due to poor interfaces
- Learn from these and fix them!

“You know, it’s really dumb to keep this right next to the cereal. ... In fact, I don’t know why we even keep this stuff around in the first place.”

Performance

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity - W.A. Wulf

Perceived performance is what really matters

- Is the system getting the job done or not?
- Function of resources, efficiency, scope, etc.

Most people can only effect efficiency

- That's why people like to tune their programs to make them more efficient
- But it might not be the best way to get improvement
  - People are expensive, often overloaded

But if you're going to tune a program, you might as well do a good job

Reminder: Performance assumes correctness!

- You have to make sure the program still works after you tune it

Start by understanding the problem

"Show me what part is taking all the time!"

Need tools to get reliable performance info

Several ways to acquire data

- Your OS probably has high-level tools for checking machine status
  - top, lsof, vmstat
  - Tools available vary with OS type
    - Sun Solaris: pmon, pstat, pstack
    - Linux tools: free, memalloc
- C/C++ have tools like gprof for internal program performance
- Java virtual machines can capture data at runtime

Several approaches:

- Periodic samples
  - Use the procedure stack in each sample to figure out what's being done
  - Use statistical arguments to provide profiles
  - Fast, simple
- Tracking call/return control flow
  - Captures entire behavior, even for fast programs
  - Requires instrumenting the code
  - Accurate

The data you get looks like this:

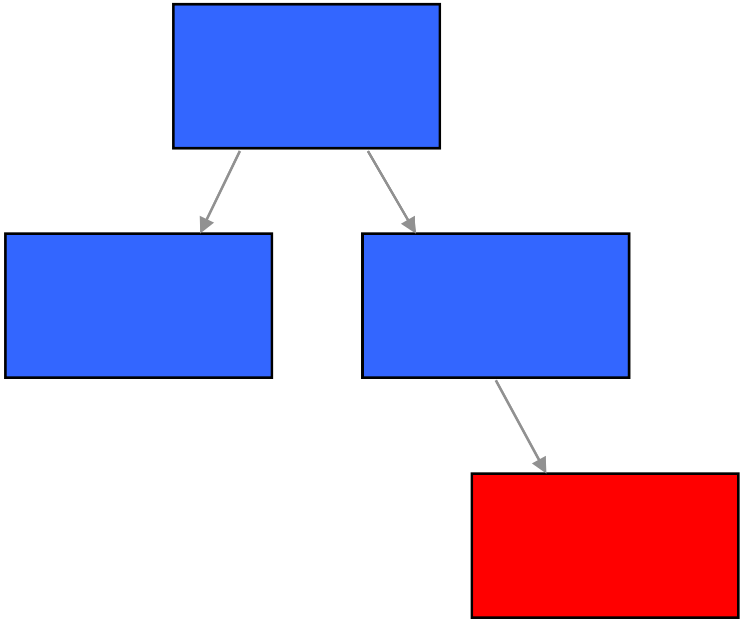
```
CPU SAMPLES BEGIN (total = 909) Sat Feb 12 13:45:46 2000
rank   self  accum   count trace method
   1 28.60% 28.60%     260    31 java/lang/StringBuffer.<init>
   2 26.51% 55.12%     241    18 java/lang/StringBuffer.<init>
   3 24.42% 79.54%     222    48 java/lang/StringBuffer.<init>
   4  4.62% 84.16%      42    21 java/lang/System.arraycopy
   5  3.96% 88.12%      36    49 java/lang/System.arraycopy
   6  3.85% 91.97%      35    36 java/lang/System.arraycopy
   7  0.66% 92.63%       6    33 com/develop/demos/TestHprof.makeStringInline
   8  0.44% 93.07%       4    47 java/lang/String.getChars
   9  0.33% 93.40%       3    23 java/lang/StringBuffer.toString
  10  0.22% 93.62%       2    25 java/lang/StringBuffer.append
  11  0.22% 93.84%       2    59 com/develop/demos/TestHprof.makeStringWithBuffer
  12  0.22% 94.06%       2    50 com/develop/demos/TestHprof.makeStringWithLocal
  13  0.22% 94.28%       2    40 java/lang/StringBuffer.toString
  14  0.22% 94.50%       2    17 com/develop/demos/TestHprof.addToCat
  15  0.22% 94.72%       2    41 java/lang/String.<init>
  16  0.22% 94.94%       2    30 java/lang/StringBuffer.append
  17  0.22% 95.16%       2     7 sun/misc/URLClassPath$2.run
```

Now what?

Now what?

What you have: How often some function was running

What you want: “Improve this place first”



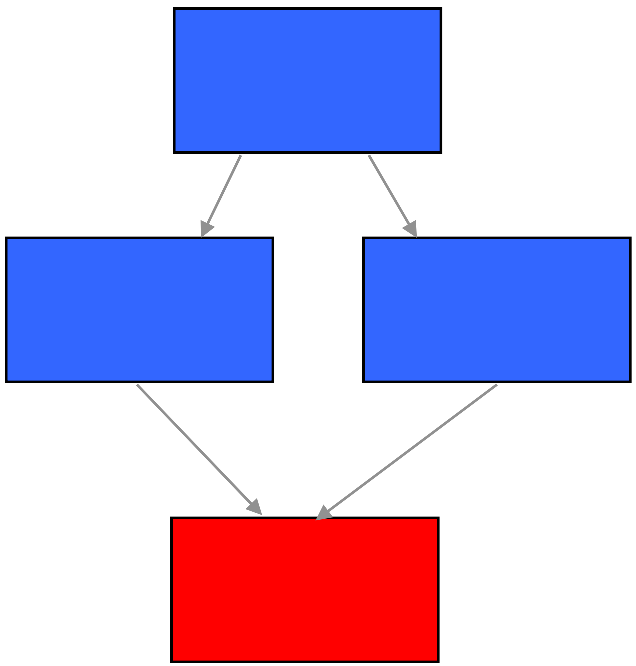
Is this asking for too much work?

Is this a poor algorithm?

Now what?

What you have: How often some function was running

What you want: "Improve this place first"



Who's responsible for all this work?

Tools to help understand performance info

Commercial performance tools tend to have powerful analysis features

- This is why people are willing to pay so much for them...

PerfAnal as an low-end example

http://developer.java.sun.com/developer/technicalArticles/Programming/perfanal/index.html

Four views of the behavior

- Top down look

  How is each routine spending its time

- Bottom up look

  Who is asking this routine to spend time?

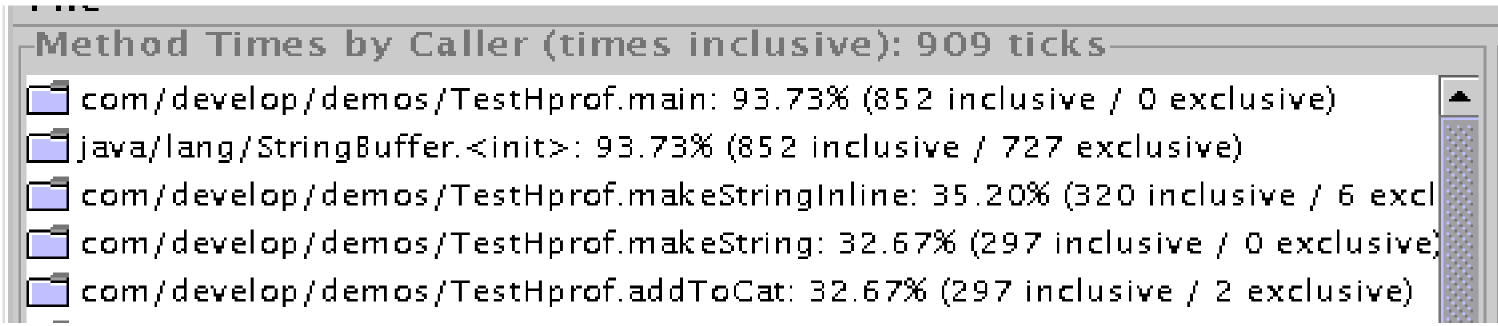
- Detail within each function by line number

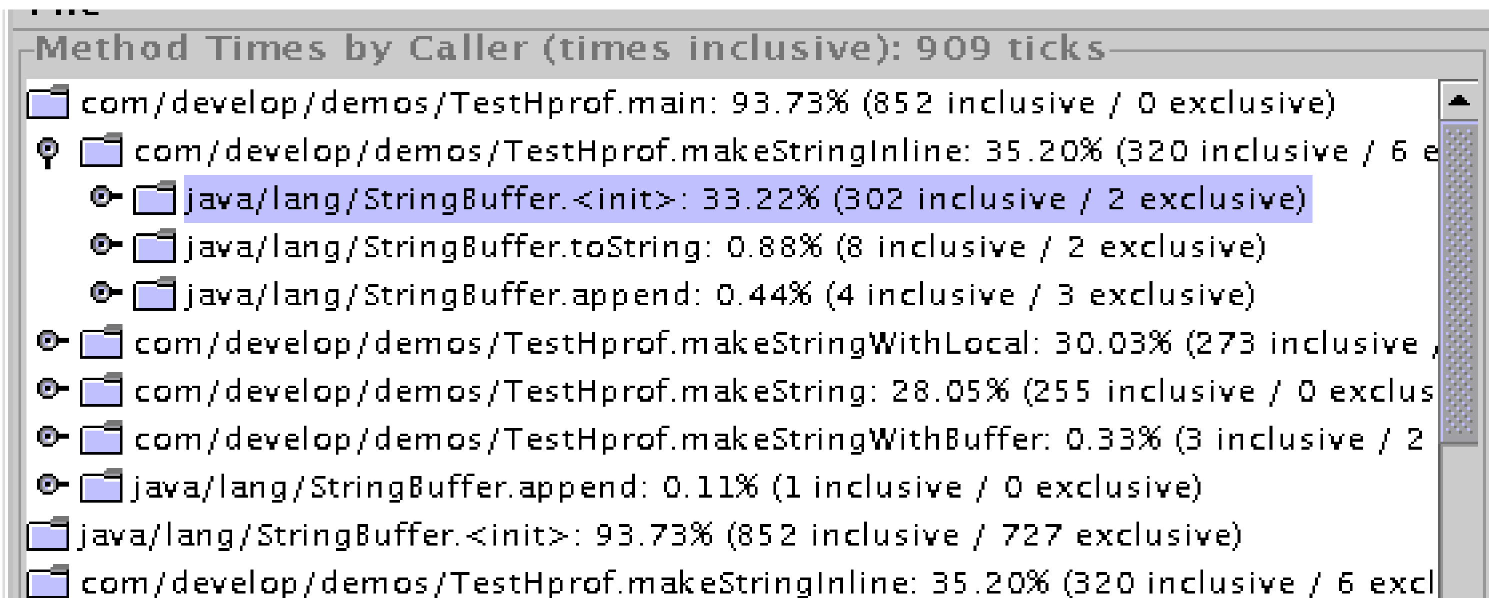
  How is time spent in each function, with/without calls to others?

  Is there just some bad code in there?

Top-down view of the program

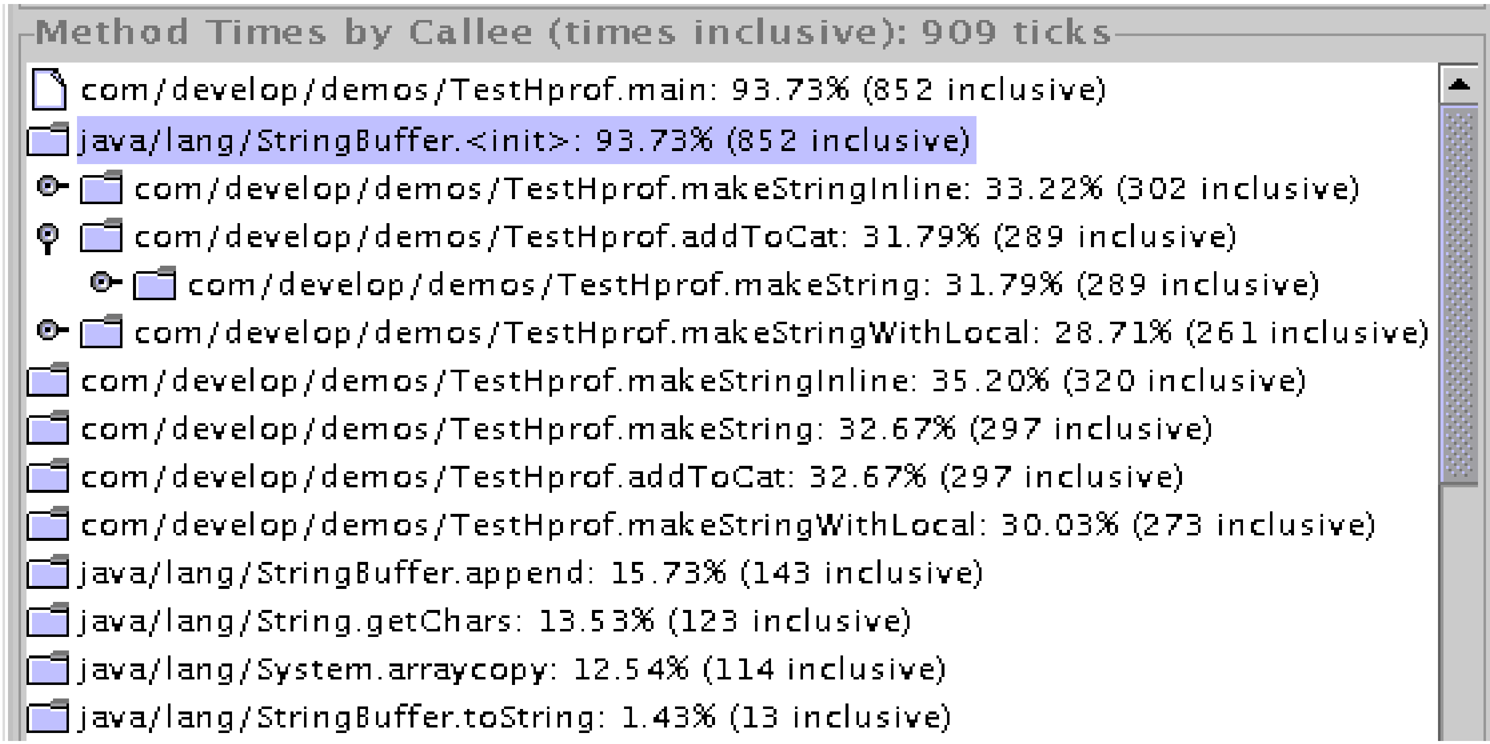
How is the routine spending its time?





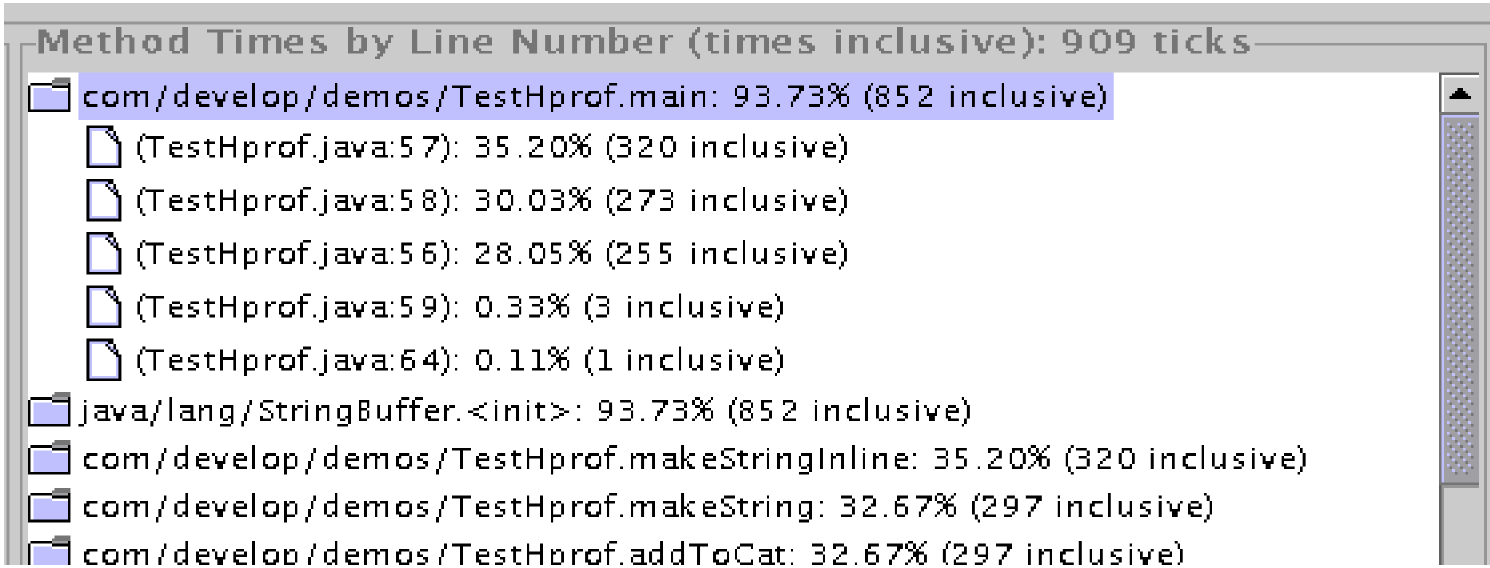
Bottom-up view

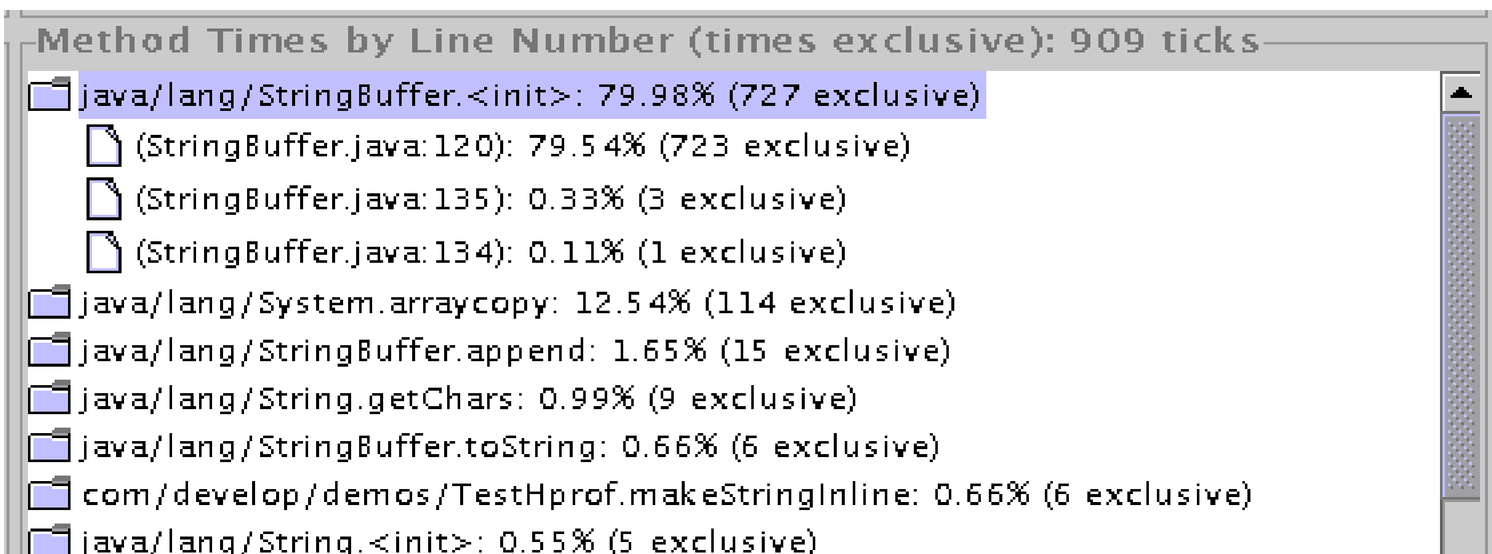
Who is asking this routine to spend time?



Even more detail…

Within a member function





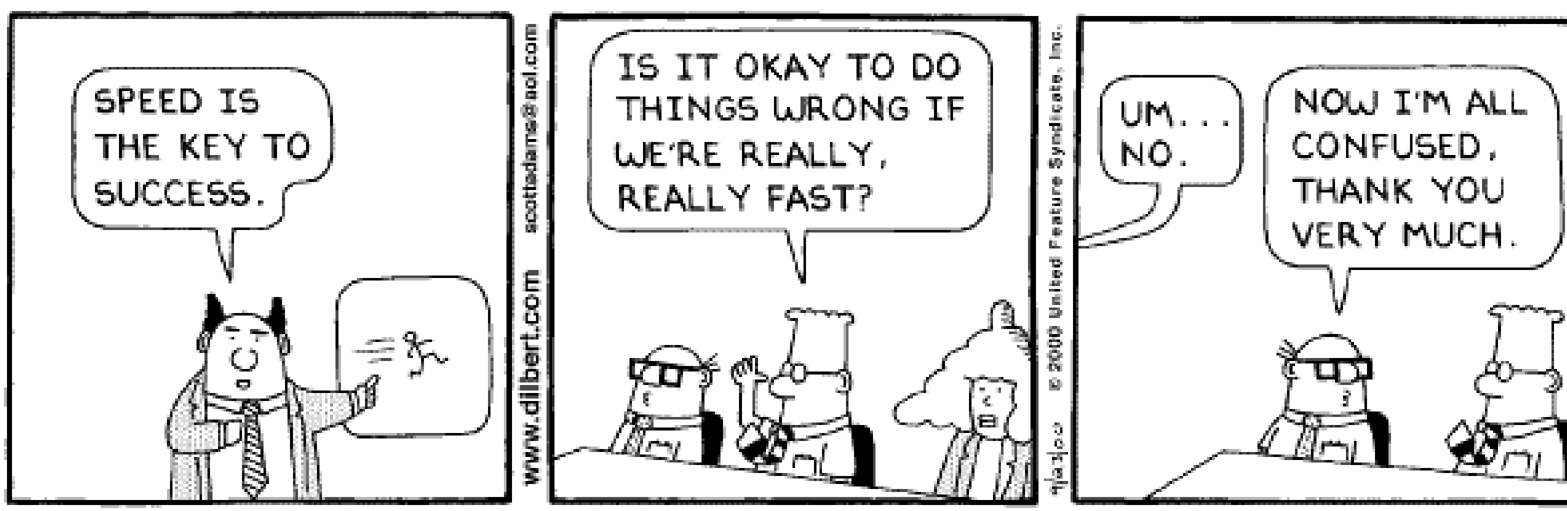
How do you use this?

Two approaches:

- Make often-used routines faster
- Call slow routines less often

But it has to stay correct!

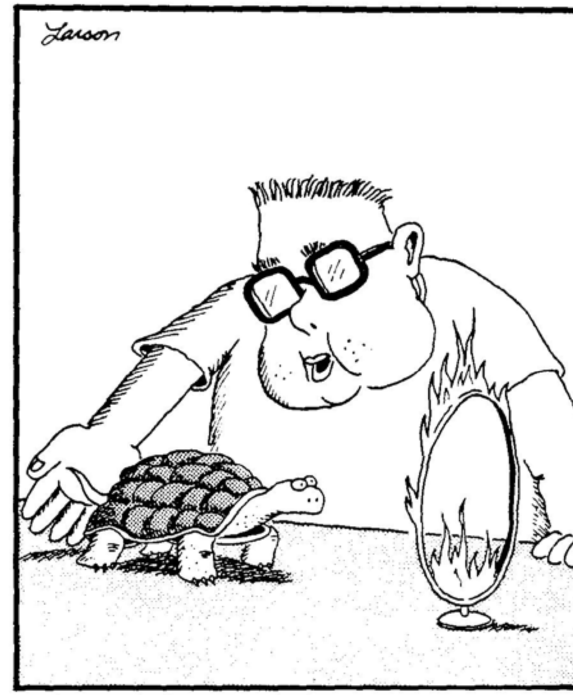
- Start by working in small steps



Sometimes you have to drop back 10 yards and punt

Not all problems will be solved with an incremental approach

- "Do we have to do this?"
- "Is there a better way to do this?"



"Through the hoop, Bob! Through the hoop!"

Traditional example: Sorting a new deck of cards

Method 1: Pattern recognition

- There are a finite number of possible arrangements
- Find which one you have, and then reorder
- 52! = 4×10^{66} so will need about $52 * 4 \times 10^{66}/2$ comparisons

Method 2: Bubble sort

- Scan through, finding the smallest number
- Then repeat, scanning through the N-1 that's left
- Cost is $O(N^2)$ "sum of numbers from 1 to N" = 52*(52+1)/2 = 1.4×10^3

Method 3: Better sorts - Shell sort, syncsort, split sort, ...

- Even for arbitrary data, better sort algorithms exist
- O(N logN) = k * 52 * 5.7 = k * 300
- For N large, important gain regardless of k
- As ideas improve, k has come down from 5 to about 1.2 => 70 calcuations

Method 4: Bin sort ("Solitaire sort")

- Use knowledge that there are 52 specific items
- Throw each card into the right bin with 52 calculations

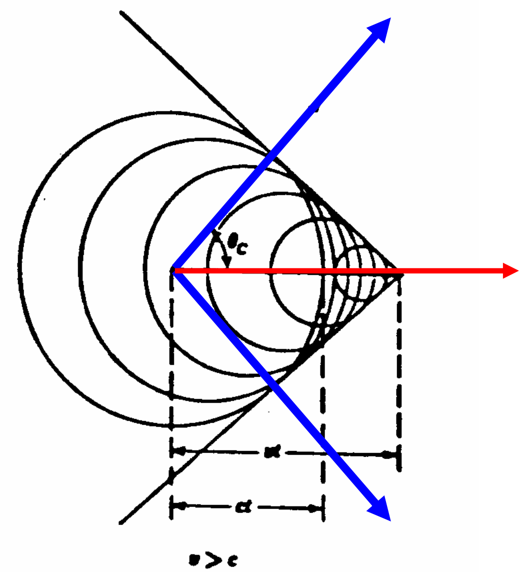
Method 5: Just look at each card in turn!

Telling pions from kaons via Cherenkov light

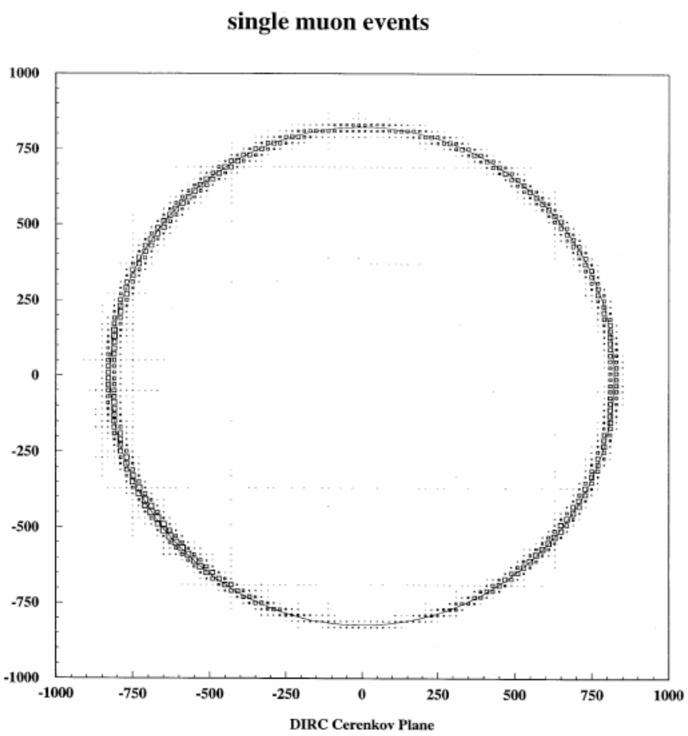
Pions & Kaons have similar interactions in matter, differ in mass

Particles moving faster than light in a medium (glass, water) emit light

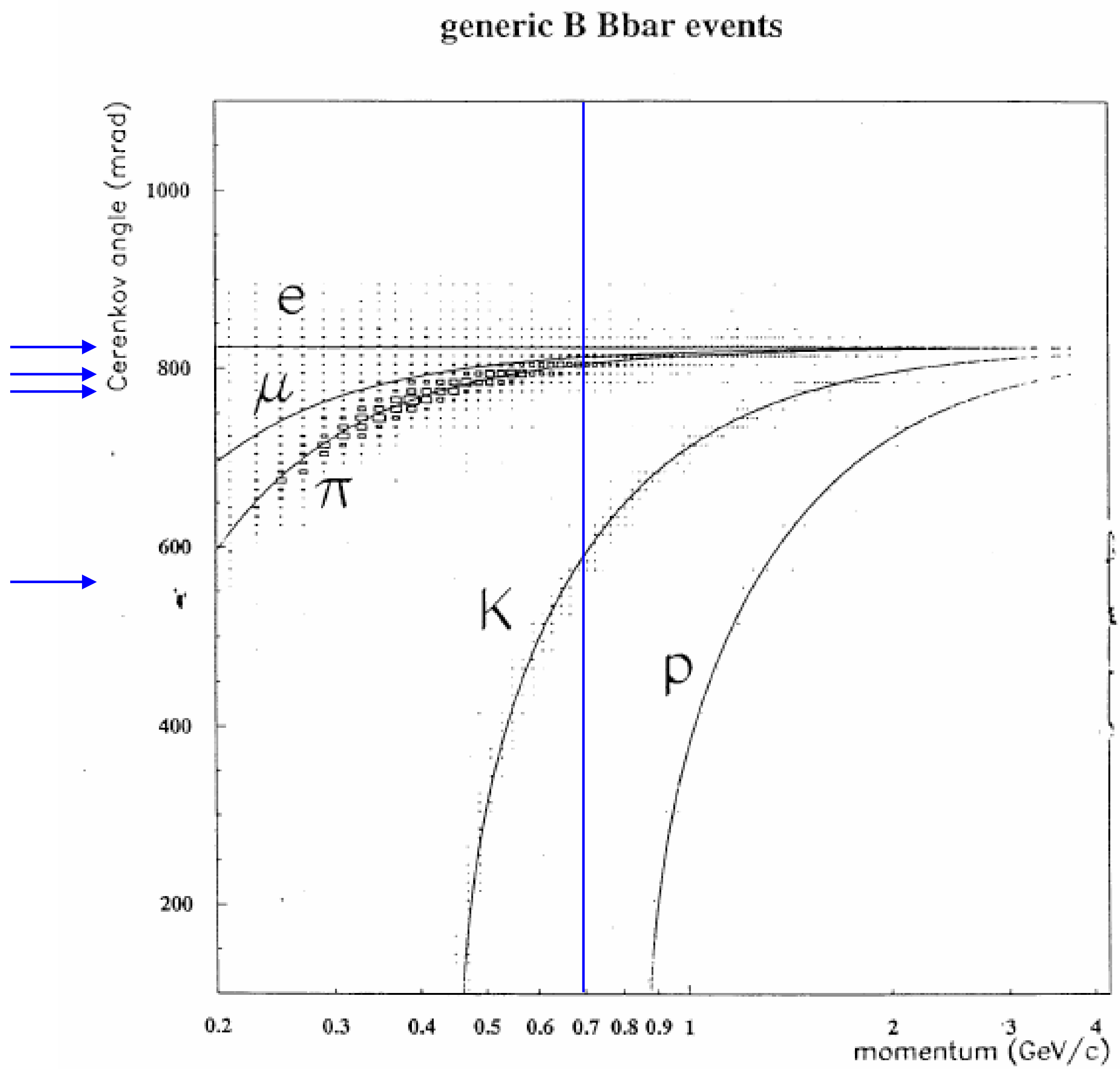
- Angle is related to velocity
- Light forms a cone



Focus it onto a plane, and you get a circle:



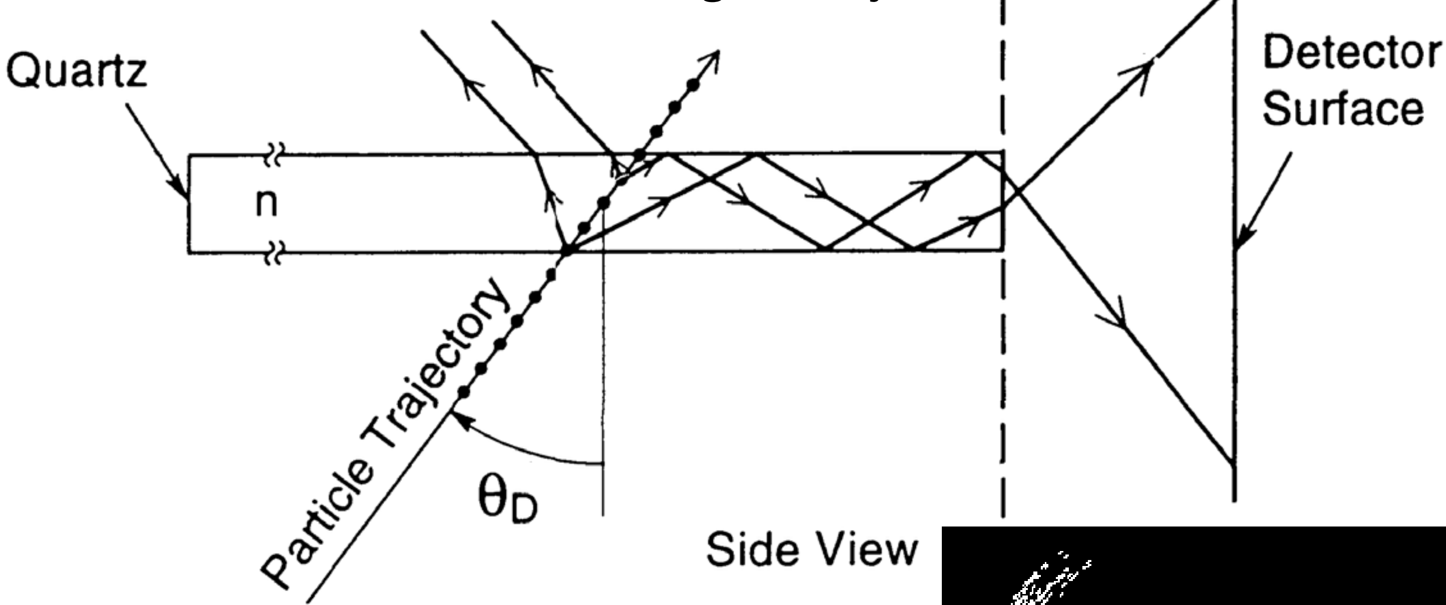
Radius of the reconstructed circle give particle type:

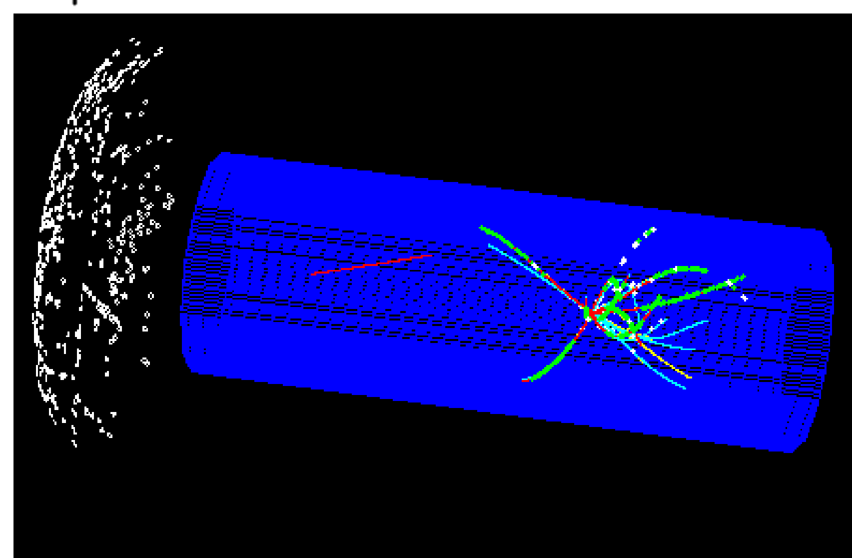


How to make this fit?

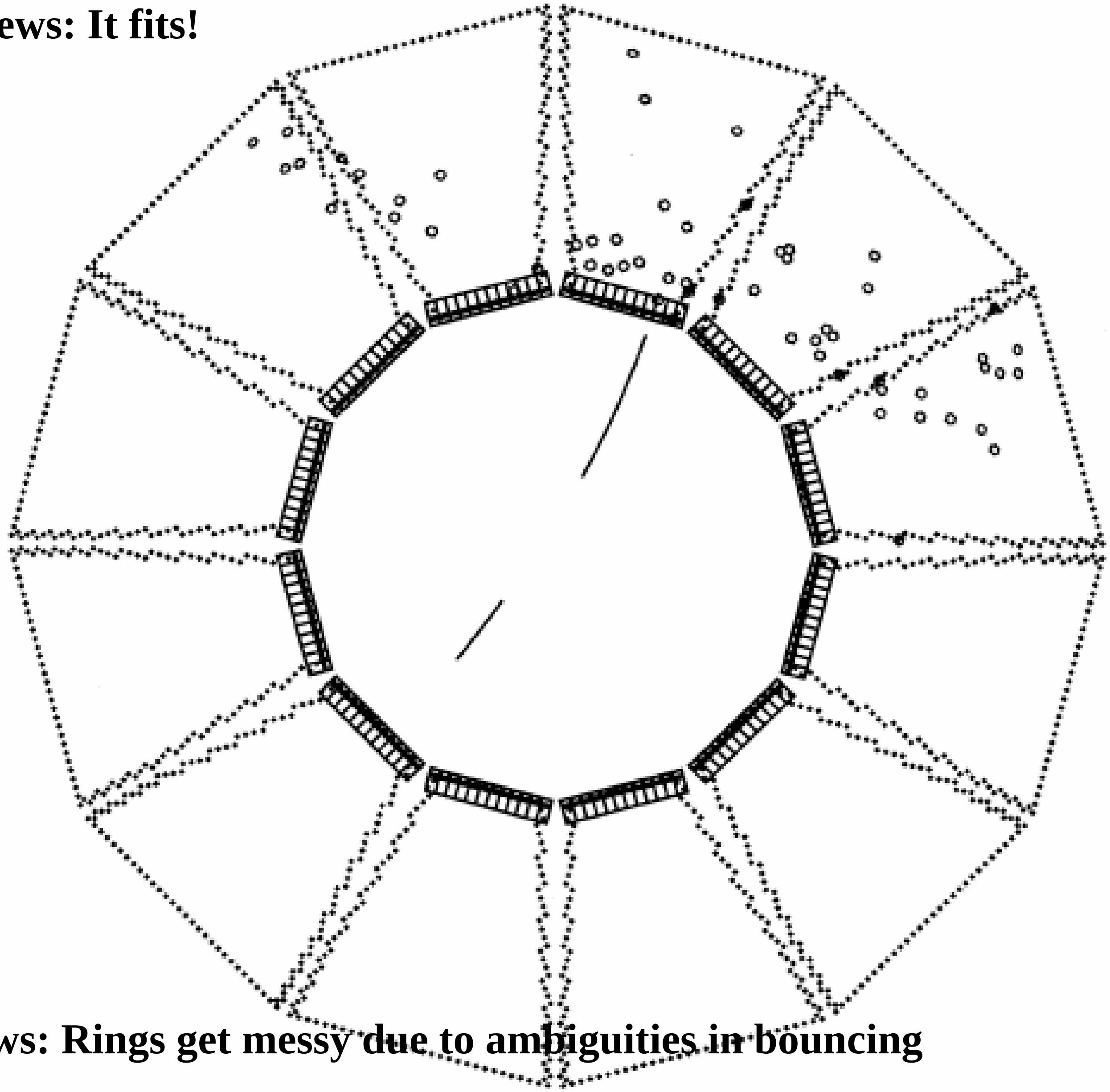
Space inside a detector is very tight, and the ring needs space to form

BaBar uses novel "DIRC" geometry:



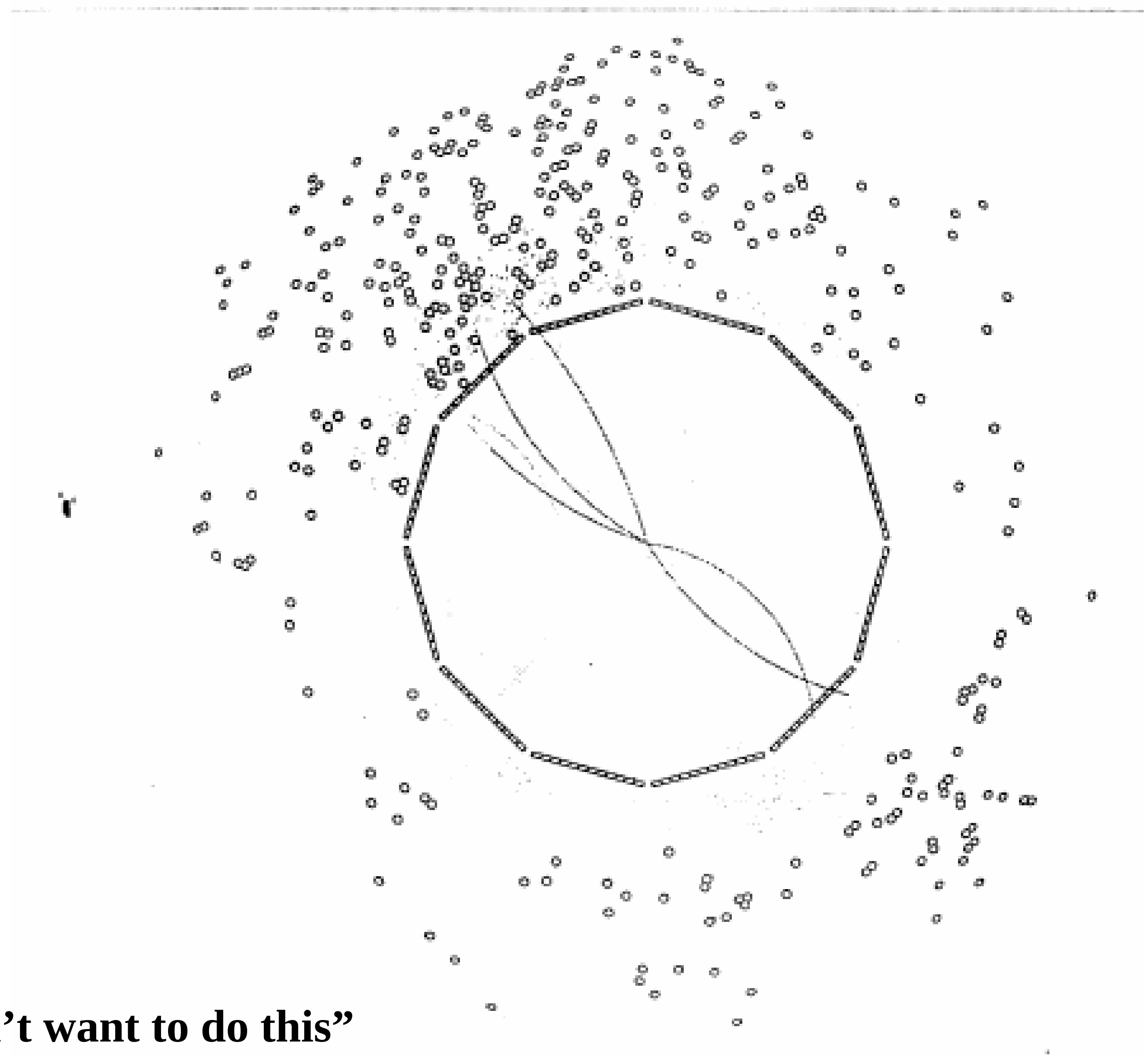


Good news: It fits!



Bad news: Rings get messy due to ambiguities in bouncing

Simple event with five charged particles:



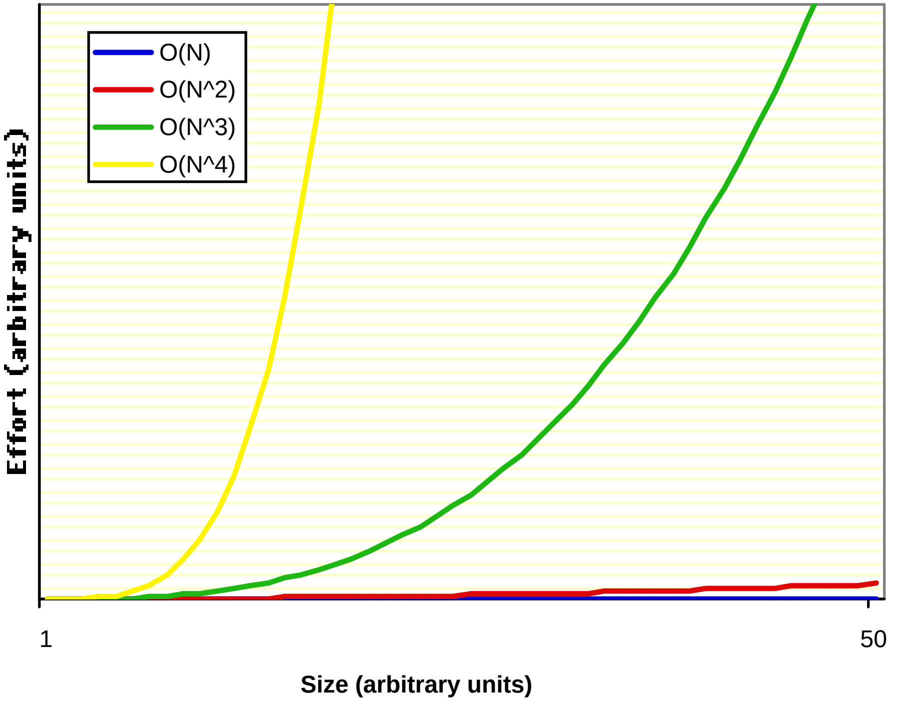
"I don't want to do this"

Why is this hard?

Brute-force circle-finding is an $O(N^4)$ problem

- Basic algorithm: Are these four points consistent with a 'circle'?

We catalog algorithms by how their cost grows with input size: O(N)



Realistic solution for DIRC? (Avoiding $O(N^4)$)

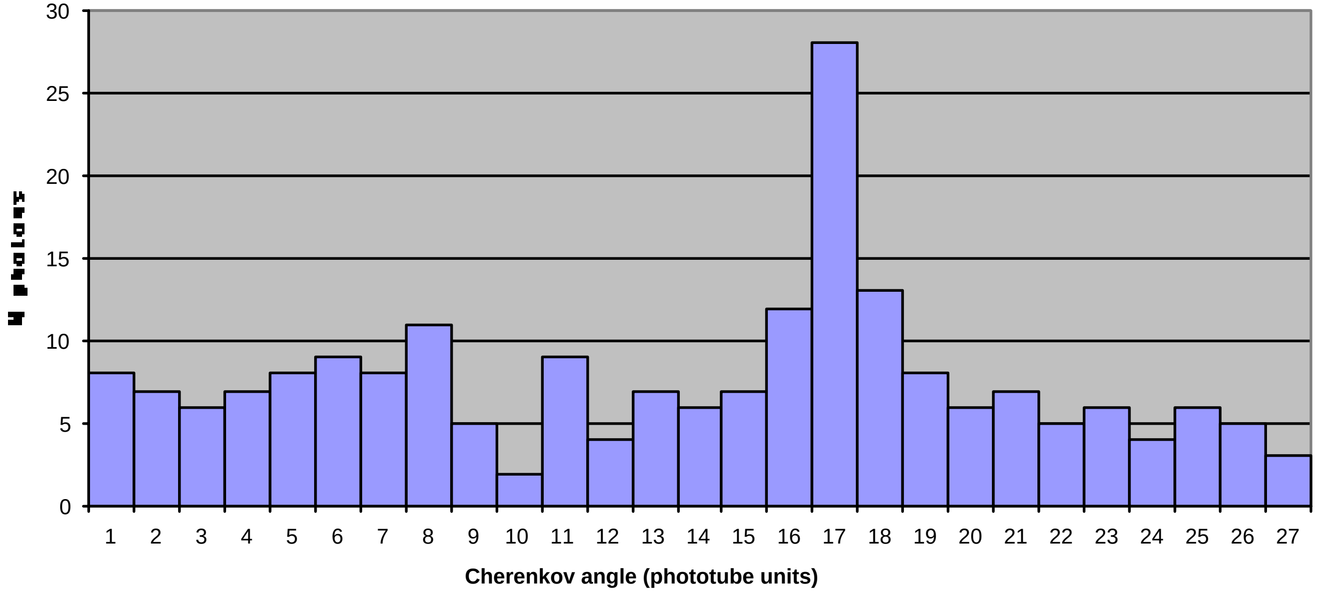
Use what you know:

- Have track trajectories, know position and angle in DIRC bars
- All photons from a single track will have the same angle w.r.t. track

  No reason to expect that for photons from other tracks

For each track, plot angle between track and every photon - O(N)

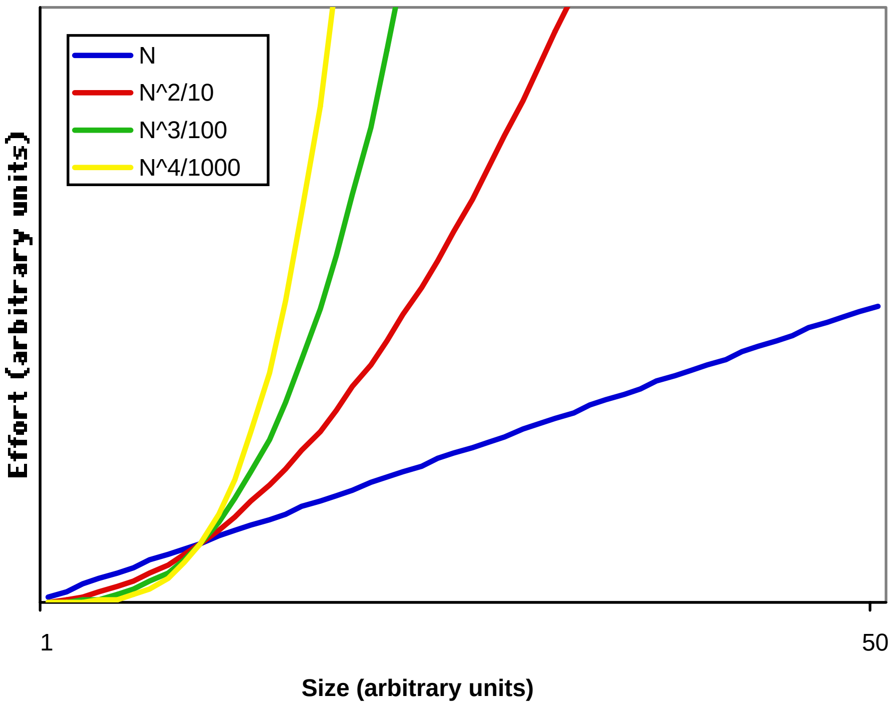
- Don't do pattern recognition with individual photons
- Instead, look for overall pattern



Not perfect, but optimal?

“But each operation is so much slower…”

How do I compare a “fast” $O(N^4)$ algorithm with a slow O(N)?



Many realistic problems deal with lots of data items

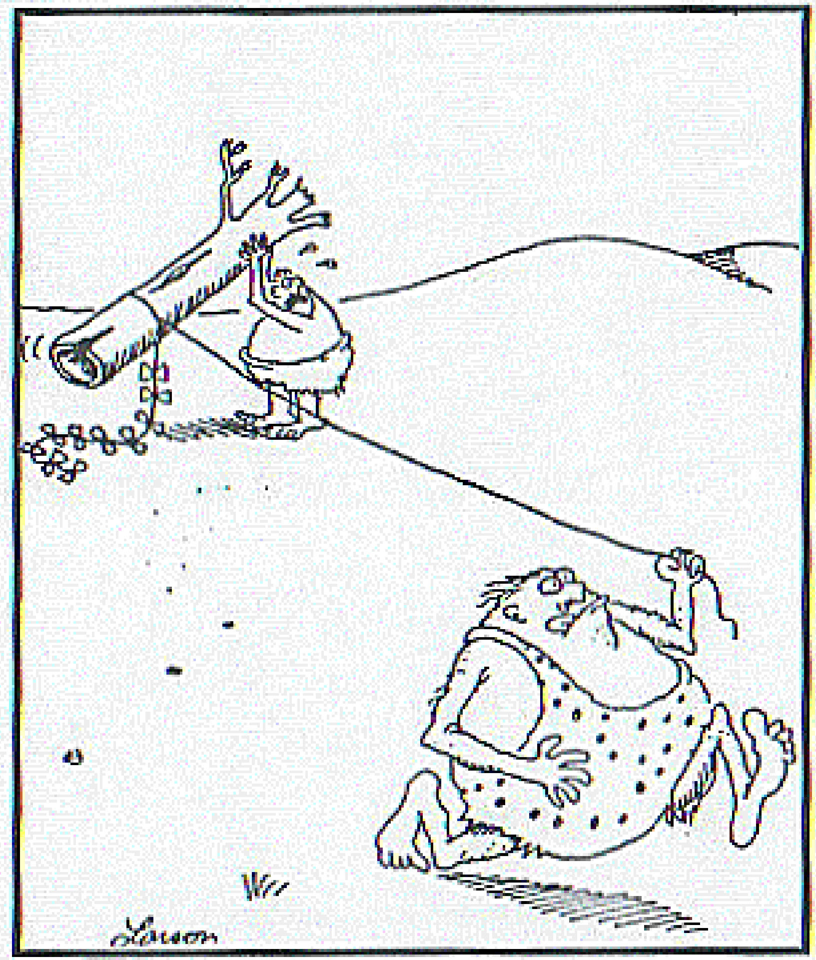
- Sharp coding is unlikely to save you a factor of 50^2 per calculation

Summary 2

Find a way of doing good work

Use tools wisely

Think about what you're doing



"Okay, Bob! Go! Go!"

Today's Exercises

If you've not used CVS, there are (optional) exercises you can start with:

1) Simple use of CVS

2) More advanced CVS, showing how conflicts are handled

Then we use a test framework, to see how it can help:

3) Demonstration of a test framework

4) Practice debugging using a test framework

And then we start working on performance:

6) Demonstration of profiling tools

7) Practice tuning a small application

Instruction sheets are available via web browser at

file:/home/jake/index.html

If you get past these, feel free to move on to tomorrow's exercises (see the instructions page)