



Performance Bugs

Jon Bentley
Avaya Labs Research

IP Telephony

Contact Centers

Unified Communication

Services

Outline

Two Performance Bugs

A bug report

A one-line error

Perspective on Performance Bugs

Tales

Lessons

A Subtheme: Software Consulting

Economics 1.01: Buying in Bulk

Unit Cost

Gasoline is \$3.09/gallon

Economy of Scale

Buy 12, get 1 free (baker's dozen)

Fast food soft drink: 20 oz: \$1.59; 32 oz: \$1.69; 44 oz: \$1.79

Diseconomy of Scale

Price of lobster

The bigger the lobster, the greater the price per pound

1–1¼: \$8.95/lb; 2–3: \$12.50/lb

Bill Engvall's "I'm Stupid" signs: \$1 each or 2 for \$5

US income taxes

The more you make, the higher the tax rate

Search area as a function of time

1. A Great Bug Report

We [Wilks & Becker] found that `qsort` is unbearably slow on “organ-pipe” inputs like “0123443210”:

```
main(int argc, char **argv)
{
    int n=atoi(argv[1]), i, x[100000];
    for (i = 0; i < n; i++)
        x[i] = i;
    for ( ; i < 2*n ; i++)
        x[i] = 2*n-i-1;
    qsort(x, 2*n, sizeof(int), intcmp);
}
```

(Continued ...)

1. Wilks and Becker, Cont.

Here are the timings:

```
$ time a.out 2000
```

```
real    5.85s
```

```
$ time a.out 4000
```

```
real   21.65s
```

```
$ time a.out 8000
```

```
real   85.11s
```

This is clearly quadratic behavior – each time we double the input size, the run time goes up by a factor of four.

A simple experiment establishes that a sort that should be $O(n \log n)$ is in fact quadratic

1. The Big Picture

Stages of Debugging

Original user thought that S had broken

His program “just stopped running”

(It would have finished in about a week)

Maintainers (Wilks & Becker) hunted down the real issue

Which piece is broken? The sort

Essential nature of input? “organ-pipe”: \wedge

A bug report

Next Step: Fix the sort

A Delicate Discussion

Petroski, *To Engineer is Human*, 1985, p. vii

“The lessons learned from those disasters can do more to advance engineering knowledge than all the successful machines and structures in the world.”

On the Pedigree of Bugs

I have worked as a programmer for over 40 years

Xerox PARC, SLAC, CMU, IBM, Bell Labs, AT&T, Avaya, ...

“The Bug with a Thousand Faces”

Everyone makes mistakes

Excellent organizations learn from them

When the fault is mine, I’ ll take the blame

2. Some Familiar Code

A Common Schema

```
count = 0;  
// ... much later, and iterated ...  
if (++count > 100) {  
    // expensive operation  
    count = 0;  
}
```

Potential Performance Problems?

What possible minor coding mistakes could be made?

What impact would they have?

2. Three Potential Performance Problems

Missing Initialization

```
count = 0; ...
if (++count > 100) {
    // expensive
    count = 0;
}
```

Impact?

$n/100 \rightarrow$

Initially positive:
little change

Initially $-\infty$: 0

Missing Increment

```
count = 0; ...
if (++count > 100) {
    // expensive
    count = 0;
}
```

0

Missing Reset

```
count = 0; ...
if (++count > 100) {
    // expensive
count = 0;
}
```

$n-100$

2. A Big System

E-mail to a Performance Team

I have found what I think is a bug. Search for the variable **LocalQCnt** and you'll find three references:

```
dup.p nspace.c <global>    85 int LocalQCnt = 0;  
dup.p queue.c <global>    17 extern int LocalQCnt;  
dup.p queue.c nd_lcl_q    173 if ((LocalQCnt++) > 200)
```

It is initialized to zero, incremented in `nd_lcl_q`, but *never reset*. After the 200th page fault, the test in `nd_lcl_q` is thereafter *always true*. This means that every time we need a new buffer, we *always flush* the queue.

Performance Impact

$N / 200 \rightarrow N - 200$

Doubles or triples page faults

Increases CPU utilization by about 50%

A Performance Bug

is a minor glitch

(such as an atypical input¹ or one missing source line²)

that does not alter the correctness of a program

(so it is not revealed by functional testing^{1,2})

but does cause the program to consume excessive resources

(such as CPU time^{1,2} or page faults²)

Dimensions of Performance Bugs

What Effect Do They Have?

Accretive – death by a thousand paper cuts²

Catastrophic – sudden and obvious¹

How Are They Discovered?

In the field, after they bite¹

By perusing code², before or after they bite

What Causes Them?

Unforeseen cases¹

Little coding boo-boos²

How Are They Conclusively Identified?

Tiny, well-structured experiments¹

Analysis²

A Survey of Problems

Sorting Problems

Q: But I don't care about sorting!

A: But you know about it, and it illustrates deep truths

And I can speak from (sometimes bitter) personal experience

System Issues

Caching, I/O speed, memory allocation

Some Big Systems

Personal Performance Bugs

Engineering a New Sort Function

Survey Existing Code

A dozen system sorts all traced to three original programs

Each was easily driven to quadratic time

E.g., an array of random 0s and 1s

Each had many similar performance bugs

3. Our New Code

Consistently $N \lg N$ in all of our tests

One user's response: It is way too slow

It takes $N \lg N$ time on my input

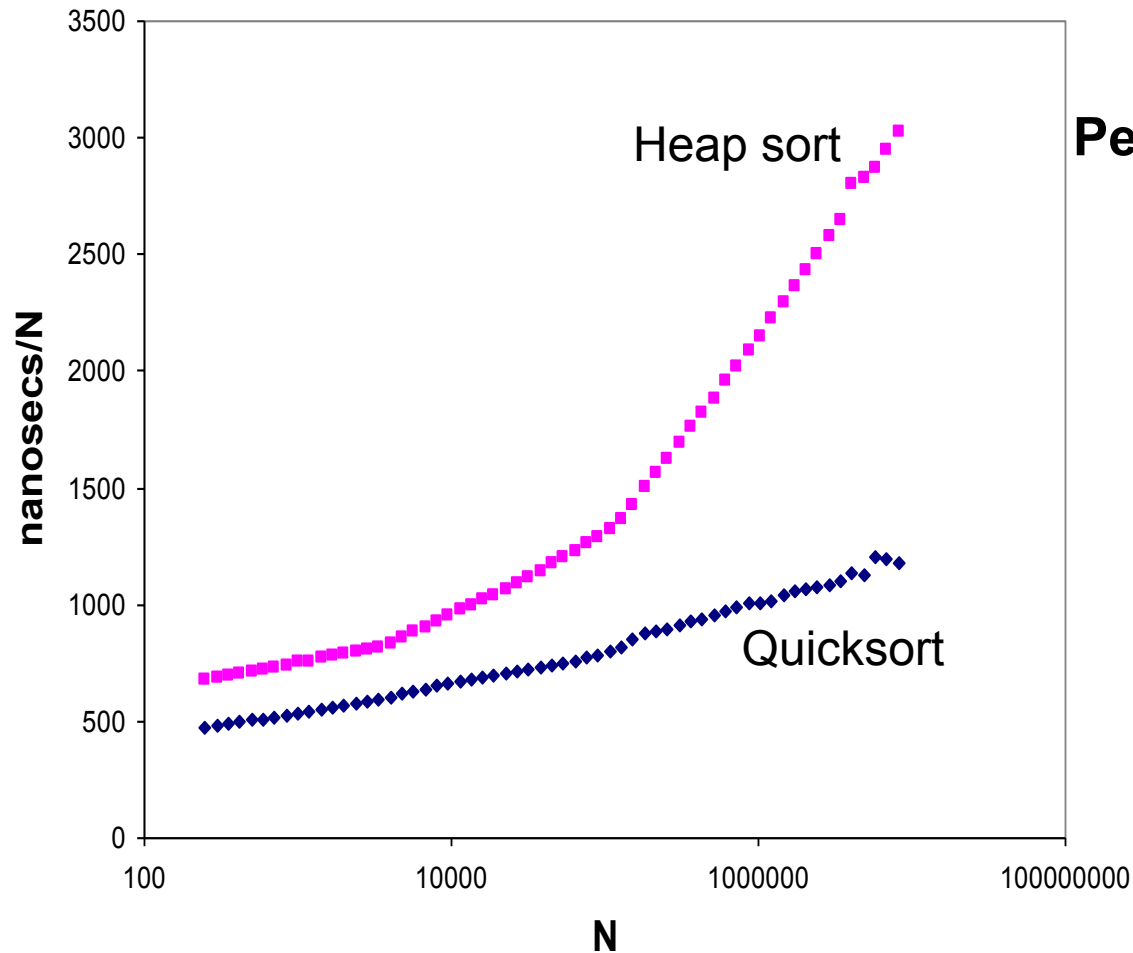
Its predecessor took just N time for my (all equal) inputs

A performance "curio" – implicitly specified by the previous implementation

4. How to implement sort – Quicksort or Heapsort?

4. CPU Times for Sorting

Sorting -- P II 400



Performance Bugs Lurking?
Experiments in one range
do not extrapolate linearly
to other ranges
Caching has a huge
influence

5. Knuth's Disk Sort

A New Disk Sort Utility

Passes all correctness tests

Takes twice as long as he predicted

Next Steps

Check and recheck the math

Simple experiments to measure the disk speed

Result: due to an OS bug, the disks had been running at half their advertised speed for over a year

Cause: “optimize” the seek to arrive just before the record rotates under the head, and miss by a little

Result of the Bug Fix

His sort ran twice as fast, just as predicted

Many other programs were faster, too

6. Another `qsort` Bug

A Bug Report

Our new sort code caused our system to crash. The maximum number of items to sort is 512. But that sort is recursive, and when sorting 512 entries, it ran out of process stack space. So we removed it.

We might want to put the sorting back. Do you know of another sort algorithm that can sort up to 512 32-bit integers very fast but without using recursion?

An Embarrassing Clarification

They were using *my* sort function

Two Separable Issues

Is the sort really guilty?

Do I know another algorithm?

6. My Response

Good news – I have 30 lines of code with worst-case time that is faster than your current sort.

But first, let's make sure that recursion is the culprit. I've attached a little recursive program that allocates about the same amount of stack memory as your sort. Here is its output on my system:

```
$ gcc recdepth.c; a.out
Passed n=10
Passed n=100
Passed n=1000
Passed n=10000
Passed n=100000
Segmentation Fault (coredump)
```

It runs at $n=10^5$, and dies at one million. A binary search could locate the exact failure point.

I suggest that you do a similar experiment. Put code like this at the place where you removed the sort, and you can find out how deep your recursion can go.

6. A Recursion Depth Tester

```
/* Sum 1+2+...+n recursively
   Use extra memory to simulate stack
   size (and fool smart compilers)
*/
int recsum(int n)
{ int i, sum, x[MEMSIZE];
  if (n == 0) return 0;
  for (i = 0; i < MEMSIZE; i++)
    x[i] = n;
  sum = 0;
  for (i = 0; i < MEMSIZE; i++)
    sum += x[i];
  return sum + recsum(n-1);
}
```

Use memory
Dink around

The Power of Profiling (And non-Performance Bugs)

Kernighan on Awk

Statement-count profile

Almost all counts were four decimal digits or fewer

Six lines of initialization code had a count near a million

Fix: only re-initialize as many as you used last time

Not a performance bug: young software tends to have time spikes

An OS Performance Group

Time profiling shows half the time in one tiny loop

Rewriting the hot spot in microcode gives $\times 10$ speedup

No change in system throughput, though

They had optimized the idle loop

Not a performance bug: a well-deserved spike

7. Profiling A Big System

A Prototype Page Profiler

“Frequent flyers” among 10^7 page faults

count	page addr	instr addr
764210	08fbb6f0	086fe958
206058	0e6fc064	088cb2c9
204235	085b1822	08081372
203168	084decc4	0809e388
181556	0e6fd114	088c5541
179481	0e232028	086ff492

Observations

About 7½% of the faults come from that one instruction

Second place is only about 2%

Time to check that instruction

7. The Expensive Piece

The Code

```
/* Do not deliver the following lines ENABLED: these
 * are ONLY used for automated unit testing.
 */
#if DY_H842_AFQ_TEST IS_ENABLED
    H842_test_phase = 0;
#endif
```

The MR

Disable DY_H842_AFQ_TEST. As the comment indicates, it is only for automated unit testing and should not be enabled. Additionally, it causes code to be executed unnecessarily. Another side effect is that it touches test flags that are stored in shadowed memory causing numerous, unnecessary page faults.

This change reduces the number of page faults and the amount of data transmitted by about 7.5%

Perspective on Performance Bugs

Causes

Big Systems

Two tiny programming goofs, each just one line of code^{2,7}

One problem with stack space⁵

Sorting

Different nature of input

Becker and Wilks' s "pipe organ"¹

Duff' s duplicates³

User expectations

Implicit contracts – Duff: "you can' t make it slower"³

Underlying system behavior: Caching⁴, disk speed⁵

Cures

Awareness, general performance testing^{4,5,7}, definitive experiments^{1,5,6}

Remainder of the Talk

More stories, with an emphasis on causes and cures

8. Memory Allocation

Classic – Memory Leaks

The program runs out of memory way too early

My Slow Program

The code is surprisingly and painfully slow

Profiling shows that the lion's share of the time goes to allocation

After days of abstraction and experiments, distill the problem to its essence

8. A Definitive Experiment

The Code

```
void main(int argc, char *argv[])
{
    int n = atoi(argv[1]);
    while (n-- > 0)
        malloc(16);
}
```

The Data

```
$ time a.out 50000
1.8u
$ time a.out 100000
8.5u
$ time a.out 200000
38.3u
$
```

Doubling input quadruples time, so it is quadratic

8. Why Is It Quadratic?

My First Assumption

I ran to gloat with my friend over a sloppy design

Doug McIlroy's Response

"That's my code!"

His reasoning 15 years earlier

Machine has 64K words of memory

Code runs in time $c_1N + c_2N^2$

The second term is always negligible

Tune so that c_1 (always the dominant term) is small

What happens 15 years later?

Moore's law implies growth of a factor of 1000

Soul Searching

How will the code you write today fare in 15 years?

Bridges and Software

Ressler on “Great Structures”

Teaching Company, 2011

Bear Mountain Bridge

Main span 1632 feet

**World’s longest suspension
bridge, 1924–1926**



“Do-Overs”: much easier for software than bridges

But What If?

One scaled a bridge design by a factor of 1000?

Leaving in one “test rivet” caused that rivet to weigh 7½% as much as the rest of the structure put together?

Leaving out one strand of wire caused the main cable weight to increase by a factor of 200?

Humility about the fragility of software

9. A Notification System

Essence: Mix and match phone, voice mail, text, e-mail, IM, ...

How long to send e-mail to N recipients?

<u>N</u>	<u>Seconds</u>
1000	11.4
2000	39.6
4000	167.1

Quadratic once again

29 hours for $N=100,000$

Why?

Sending the same piece of e-mail twice is wasteful

Therefore keep a list of recipients, and search it for each

How to find such bugs?

Tiny performance tests along critical dimensions

Force accretive bugs to become catastrophic

Personal Performance Bugs

10. Change the max function to a macro

Sometimes a speedup of 2

In a recursive function, a slowdown of 10,000

Details in *Programming Pearls*, 1999, Sec. 9.2

11. Hash table size

8191: List size is 1.92

8192: List size is 3510

A disaster waiting to happen!

12. A clever caching scheme

N \ P	1	100	2000	10000
10000	100.0	98.6	91.3	89.5
9000	100.0	106.7	91.5	193.2
7000	100.0	104.0	211.5	2463.4

10.5% better!

mixed

25 times worse!

Review of Performance Bugs

1. Changing the nature of inputs to be sorted
2. Forgetting one reset to zero
3. A new sort runs in optimal time and is too slow
4. Different speeds in different memory hierarchies
5. Knuth's double-time disk sort
6. Excessive stack space kills the system
7. A unit test feature was enabled in production
8. Memory management grows by a factor of 1000
9. An e-mail system caches its recipient list
10. Change a function to a macro
11. 8191 → 8192
12. Clever new cache

Some Causes of Performance Bugs

Changes to the Code

Tiny coding goofs: missing reset², uncommented line⁷

Small changes with unforeseen big impact^{5,10,11,12}

New Inputs

Unexpected size: McIlroy's malloc⁸, sort with caching⁴

Different nature of input

Becker and Wilks' s "pipe organ"¹, cache access patterns⁴

User Expectations

Inadequate performance specifications

Implicit contracts: Duff's "you can't make it slower"³

Underlying System Behavior

Caching⁴; memory allocation⁸; disk speed⁵; sort performance¹

Premature Optimization^{1,3,5,9}

Attacking Performance Bugs

Awareness

Specification

Design

Coding

Testing

Performance Models

Profiling

Ongoing Monitoring

Identifying the Culprit

Certifying the Problem

Repairing

Performance Testing

Test Both Components and Overall System

Monitoring and Profiling Tools

Count anything and everything

Microseconds, kilobytes, pages, critical events, ...

Automate tests, analyses and meta-analyses

Experiments

Exploratory – *Where* does the time go?

Page profiles, e-mail system

Confirmatory – Does it *really* go there?

Memory allocation, sorting time and space, e-mail

Maintenance – Did I introduce any new bugs?

Concrete Proposals

Catalog Recent Performance Bugs

Search for patterns

Perform root cause analyses

Mechanically scan for missing resets

Were other tests were enabled? Would this help?

```
#if UNIT_TEST && DY_H842_AFQ_TEST IS_ENABLED
```

Add a time measurement to every single functional test

Test performance along critical dimensions

Routinely gather profiles and perform meta-analyses

Insert run-time checks into delicate structures