Performance Engineering in a Legacy System

Jon Bentley, Duffy Boyle, P. Krishnan, John Meiners
Outline

Context: A Big System
The Cache
Searching the Cache
  Hash tables and LRU
Understanding the Cache
  Finding patterns and exploiting them
Data Compression
Perspective

Theme: Adventures in consulting
The Context

A Telecom System

10,000,000+ source lines, designed several decades earlier
$300+ million in annual sales

Duplicated Hardware

Data in “shadowed” memory on the active processor is continuously copied to the standby
Goal: Make (future) SW Dup as fast as (current) HW Dup

Approaches

Copy less data
Make the duplication process more efficient
Anything that works: “No rules in a knife fight”
The Big Win: The Cache

Without a Cache
Each 4,096-byte page that is written is sent across

With the Cache
The cache stores a set of (say) 10,000 pages
At each page change, the cache is searched
   If it is not found, the page is transmitted in its entirety
   If the page is in the cache
      No changes: Nothing transmitted
      Few changes: Changes transmitted
      Many changes: Entire page transmitted

The Result: Data Compression on Steroids
Transmitting differences squeezes data by a factor of 60
The Page Cache

Research,

My cache holds 10,000 entries – that number might change. A linked list stores the entries in order from MRU to LRU. I start a search at the front. If I find it, I move it up to the head, and it is the new MRU entry. If I don’t find it, it goes at the front, and the LRU entry is dropped.

Each page is identified by a \{client number, page address\} key. This linear search is likely to find the entry early – we look at only 130 entries on the average. But I think we can do better. Any suggestions for a new algorithm?

– Development
Is LRU The Best Choice?

D, Before I go further, I wanted to make sure that LRU is the best choice. I thought it was, but I wanted to hear from an expert, so I just had a long talk with ___.

Fortunately, the title of his Ph.D. thesis is “Online Prediction Algorithms for Databases and Operating Systems”. His opinion was that your approach is rock-solid. After a survey of the literature, he was able to show that my proposal of mixing in counts along with LRU was all wet. The literature says that what you are doing is indeed best. – R

Always take time to ask “is this the right problem?”
Cache Effectiveness

How effective is the cache currently?
What happens if we change the cache size?

Ways to answer the question

Theoretically

Run the real system
About ½ hour per run

“Essential Data”: In vitro experiments on in vivo data
10,000,000 (addr, client) pairs in 40 megabytes
A simple (to start with) program for experiments
(This considers only the cache hit rate;
later experiments study compression rates.)
### Effect of Cache Size

**Experiments on** $10^7$ **pairs**

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Misses</th>
<th>Average Comps</th>
<th>Seconds (\sim \text{comps} + 5)</th>
<th>Hit Rate %</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8625989</td>
<td>10.2</td>
<td>15</td>
<td>13.7</td>
</tr>
<tr>
<td>20</td>
<td>6237998</td>
<td>17.2</td>
<td>22</td>
<td>37.6</td>
</tr>
<tr>
<td>50</td>
<td>3528979</td>
<td>32.2</td>
<td>37</td>
<td>64.7</td>
</tr>
<tr>
<td>100</td>
<td>1749099</td>
<td>44.5</td>
<td>49</td>
<td>82.5</td>
</tr>
<tr>
<td>200</td>
<td>996614</td>
<td>56.9</td>
<td>61</td>
<td>90.0</td>
</tr>
<tr>
<td>500</td>
<td>502645</td>
<td>78.4</td>
<td>83</td>
<td>95.0</td>
</tr>
<tr>
<td>1000</td>
<td>196531</td>
<td>95.3</td>
<td>99</td>
<td>98.0</td>
</tr>
<tr>
<td>2000</td>
<td>80625</td>
<td>107.9</td>
<td>112</td>
<td>99.2</td>
</tr>
<tr>
<td>5000</td>
<td>48024</td>
<td>124.1</td>
<td>128</td>
<td>99.5</td>
</tr>
<tr>
<td>10000</td>
<td>11746</td>
<td>130.0</td>
<td>134</td>
<td>99.9</td>
</tr>
<tr>
<td>20000</td>
<td>10471</td>
<td>130.0</td>
<td>135</td>
<td>99.9</td>
</tr>
</tbody>
</table>

Smooth, well-behaved functions
A day *in vivo* or 15 minutes *in vitro*

10,471 distinct elements
#include <stdio.h>
define MAXSIZE 100000
#define FILESIZE 10000000

int main(int argc, char *argv[]) 
{   FILE *fd = fopen("clientaddr.bin","r");
    int i, line, inserts = 0;
    int tablesiz = atoi(argv[1]);
    unsigned char client_id;
    unsigned long address, newaddr, key;
    unsigned long x[MAXSIZE+1];
    int top = 0, comps, totalcomps = 0;
    for (line = 0; line < FILESIZE; line++) {
        /* read data and convert it */
        fread(&client_id,1,1,fd);
        fread(&address,4,1,fd);
        newaddr =
            address >> 24
        | (((address >> 16) & 0xff) <<  8)
        | (((address >>  8) & 0xff) << 16)
        | (((address      ) & 0xff) << 24);
        key = newaddr | client_id;
        x[top] = key;
        for (i = 0; x[i] != key; i++)
            comps = i+1;
        if (i >= top) {
            inserts++;
            if (top < tablesiz)
                top++;
        }
        for ( ; i > 0; i--)
            x[i] = x[i-1];
        x[0] = key;
        if (0) {
            printf("key=%08X comps=%d x: ",
                key, comps);
            for (i = 0; i < top; i++)
                printf(" %08X", x[i]);
            printf("\n");
        }
        totalcomps += comps;
    }
    printf("cache:%d top:%d inserts:%d \
        records:%d comps:%d avg:%8.2f\n",
        tablesiz, top, inserts,
        FILESIZE, totalcomps,
        (float) totalcomps/FILESIZE);
    return 0;
}
Searching the Cache

The Current Implementation

Sequential search in an LRU-MRU (Move-to-Front) list

How many comparisons for \( n = 10,000 \)?
  - Worst case: 10,000
  - Observed average: 130

Faster Implementations?

Sketch of One Approach

How to find the LRU entry?
  - Store use time with each element
  - How to find extreme in a dynamic set? Heap!

Use a hash table to search by \{addr, client\} key

Might give \( O(1) \) for hashing + \( O(lg n) \) for heap

Can we do better?
Simpler Approach

Combine

Hashing – for looking up the pair
Doubly Linked List – for maintaining the LRU list

How to connect the two structures?
Pointers back and forth
Sharing the same nodes
Describing the Data Structure

D, I've attached my current code (180 lines), about half of which is documentation.

I'm now sure that we can reduce the linear search down to constant expected time. The initial comments tell the strategy of the data structure, and the code itself is documented.

This program might work – it has run one test case – but I'm sure that it can be made to work. I'd like to talk about it with you before I devote much more time to polishing it. – R

R, I've looked over this code and I understand it. I can run with it from here, so there's no need for you to finish it unless you want to. – D
Program Documentation

This solution combines two data structures:

A (chained) hash table to support $O(1)$ expected-time lookup:

```
|___|___|___|___|___|___|___|___|___|___|  hashtab
|   |   |   |
 x   x   x   nodes and pointers
|   |
 x
```

A doubly linked list to maintain the MRU..LRU order

The MRU item is at the front of the list, the LRU at the back

This structure supports these operations in $O(1)$ time

- Put a new item at the front
- Delete the LRU item
- Delete a recently accessed item from its current position in the list and move it to the front

For convenience, this representation uses a sentinel:
```
-> Sentinel <-> value <-> value <-> value<- |
---------------------------------------------
```
Production Code Bloat

A Svelte Research Prototype
  180 lines: 101 NCSL + 79 comments
The Huge Development Version
  216 lines: 106 NCSL + 110 comments

D: Wow. Your code is beautiful indeed – a really nice combination of industrial-strength comments over “just enough” code. It is a delight to read. – R
Hash Function

Goal: How to implement

    NodeIndex hashfunc(int addr, short client_num)

Issues

    Table size?
    How to scramble the inputs?

A Place Holder

    /* Starter hash function only -- fix it soon */
    NodeIndex hashfunc(int a, short c)
    { return ((1+8+64)*a + (1+4+16)*c) % HASHSIZE;
    }

    D, If you use this one, it is important to make sure that HASHSIZE is prime; 9973 and 10007 are two that spring to mind. – R
    R, My HASHSIZE right now is 10000, definitely not prime. But it still seems to work adequately. – D
Hash Function

*In vitro* experiments on 10,000,000 *in vivo* pairs

Estimate search cost by $\sum_i count_i^2 / n$

<table>
<thead>
<tr>
<th></th>
<th>8191</th>
<th>8192</th>
<th>8209</th>
<th>9973</th>
<th>10000</th>
<th>10007</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a + c$</td>
<td>1.92</td>
<td>3510</td>
<td>2.06</td>
<td>1.69</td>
<td>11.5</td>
<td>1.77</td>
</tr>
<tr>
<td>$a &gt;&gt; 7 + c$</td>
<td>90</td>
<td>4196</td>
<td>90</td>
<td>90</td>
<td>102</td>
<td>90</td>
</tr>
<tr>
<td>$a &gt;&gt; 12 + c$</td>
<td>33</td>
<td>189</td>
<td>33</td>
<td>33</td>
<td>47</td>
<td>33</td>
</tr>
<tr>
<td>$a \times (64+8+1)$ + $c \times (16+4+1)$</td>
<td>1.84</td>
<td>3510</td>
<td>1.92</td>
<td>1.83</td>
<td>11.5</td>
<td>1.79</td>
</tr>
<tr>
<td>$a &gt;&gt; 7 \times (64+8+1)$ + $c \times (16+4+1)$</td>
<td>1.93</td>
<td>27.8</td>
<td>1.84</td>
<td>1.62</td>
<td>11.5</td>
<td>1.74</td>
</tr>
<tr>
<td>$a &gt;&gt; 12 \times (64+8+1)$ + $c \times (16+4+1)$</td>
<td>1.86</td>
<td>2.01</td>
<td>1.95</td>
<td>1.65</td>
<td>1.61</td>
<td>1.55</td>
</tr>
</tbody>
</table>

Belt and braces: good function *and* good table size
Watching the Cache

Goal: More Insight into Behavior
Method: Look at the Cache
   Exploit a new view of existing data
Implementation
   At intervals throughout the simulation …
   … print (position, count, value) triples
**Both Ends of the Final Cache**

### The Front – MRU

<table>
<thead>
<tr>
<th>Pos</th>
<th>Count</th>
<th>Addr/Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6994</td>
<td>084BB011</td>
</tr>
<tr>
<td>1</td>
<td>1109</td>
<td>08740011</td>
</tr>
<tr>
<td>2</td>
<td>76256</td>
<td>08127011</td>
</tr>
<tr>
<td>3</td>
<td>43925</td>
<td>085AB011</td>
</tr>
<tr>
<td>4</td>
<td>43925</td>
<td>085AA011</td>
</tr>
<tr>
<td>5</td>
<td>287081</td>
<td>08561011</td>
</tr>
<tr>
<td>6</td>
<td>287071</td>
<td>0848E011</td>
</tr>
<tr>
<td>7</td>
<td>1367</td>
<td>08090013</td>
</tr>
<tr>
<td>8</td>
<td>7007</td>
<td>17365019</td>
</tr>
<tr>
<td>9</td>
<td>9930</td>
<td>08FFD019</td>
</tr>
<tr>
<td>10</td>
<td>232914</td>
<td>0CEAF019</td>
</tr>
<tr>
<td>11</td>
<td>233120</td>
<td>0CEAB019</td>
</tr>
<tr>
<td>12</td>
<td>1757</td>
<td>14502019</td>
</tr>
<tr>
<td>13</td>
<td>104931</td>
<td>0CBF1019</td>
</tr>
<tr>
<td>14</td>
<td>45298</td>
<td>0CD3D019</td>
</tr>
<tr>
<td>15</td>
<td>64993</td>
<td>201D8019</td>
</tr>
</tbody>
</table>

### The Back – LRU

<table>
<thead>
<tr>
<th>Pos</th>
<th>Count</th>
<th>Addr/Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>125</td>
<td>16D7C019</td>
</tr>
<tr>
<td>11</td>
<td>412</td>
<td>16FC0019</td>
</tr>
<tr>
<td>12</td>
<td>124</td>
<td>16D7B019</td>
</tr>
<tr>
<td>13</td>
<td>30</td>
<td>087A1011</td>
</tr>
<tr>
<td>14</td>
<td>125</td>
<td>16D7A019</td>
</tr>
<tr>
<td>15</td>
<td>25</td>
<td>087A0011</td>
</tr>
<tr>
<td>16</td>
<td>124</td>
<td>16D79019</td>
</tr>
<tr>
<td>17</td>
<td>201</td>
<td>16FBF019</td>
</tr>
<tr>
<td>18</td>
<td>29</td>
<td>0879F011</td>
</tr>
<tr>
<td>19</td>
<td>125</td>
<td>16D78019</td>
</tr>
<tr>
<td>20</td>
<td>38</td>
<td>16D77019</td>
</tr>
<tr>
<td>21</td>
<td>30</td>
<td>0879E011</td>
</tr>
<tr>
<td>22</td>
<td>24</td>
<td>0879D011</td>
</tr>
<tr>
<td>23</td>
<td>3</td>
<td>17346019</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>17347019</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>0879C011</td>
</tr>
<tr>
<td>26</td>
<td>15</td>
<td>0879B011</td>
</tr>
</tbody>
</table>

**Paired Pages**

**Observations? Patterns?**

**Sequences**
Hacking the Cache

A 6-line change that hunts for runs decreases cache misses by about 10%, down to the minimal possible. Rarely used long sequences hog lots of cache space. The new code therefore treats the first few elements in a sequence normally, but doesn’t move later values to the front. The hope is that these rare items will fall off the end of the cache more quickly.

I do not recommend putting this into the system! This technique often increases the amount of work done, sometimes significantly. – R

The first test was misleading.
Long sequences still look suspicious!
Perspective on Problem Solving

A Piet Hein Grook

Problems worthy of attack
prove their worth by fighting back.

The Fundamental Axiom

Ain’t no horse that can’t be rode,
ain’t no cowboy can’t be threwed.
Long Ascending Sequences

Team, R has cataloged cases where consecutive pages are modified in order. These sequences all have the smell of code that is not efficiently using duplicated memory, and should be investigated.

Here’s the first such sequence, cataloged by R, identified by me, and explained by ___. Every 60 seconds, the ___ wakes up and audits its tables. It increments a “stale” counter in the ___ record. This activity causes 1657 consecutive shadowed pages to be modified. If we can move this counter out of the shadowed record and into a separate, unshadowed table, we could eliminate all of this shadowing.

R has identified four other places where 1000 or more consecutive shadowed pages are modified, and a number of others with shorter sequences.
An LRU-LFU Continuum

Observation

Average entry occurs 1000 times
Many elements are rare – 10% occur just once

Goal

Keep the benefits of LRU, while tossing out some small-count elements

A Hybrid LRU/LFU Algorithm

Use the Move-To-Front (LRU) heuristic on the entire sequence

To delete an element, delete the Least Frequently Used (LFU, or Count) heuristic among the final elements

Among how many? A parameter

1 gives LRU (Move to Front), \( N \) gives LFU (Count)
# Hybrid Performance

## Hybrid Cache Misses as a Percent of Pure LRU

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Pure LRU</th>
<th>25</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>Pure LFU</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>100.0</td>
<td>99.3</td>
<td>98.6</td>
<td>96.3</td>
<td>93.6</td>
<td>89.5</td>
<td>91.3</td>
<td>89.5</td>
</tr>
<tr>
<td>9000</td>
<td>100.0</td>
<td>107.5</td>
<td>106.7</td>
<td>103.0</td>
<td>98.6</td>
<td>95.2</td>
<td>91.5</td>
<td>193.2</td>
</tr>
<tr>
<td>7000</td>
<td>100.0</td>
<td>102.2</td>
<td>104.0</td>
<td>103.8</td>
<td>102.8</td>
<td>100.9</td>
<td>211.5</td>
<td>2463.4</td>
</tr>
<tr>
<td>5000</td>
<td>100.0</td>
<td>99.8</td>
<td>99.0</td>
<td>98.7</td>
<td>97.5</td>
<td>93.5</td>
<td>88.7</td>
<td>1584.7</td>
</tr>
<tr>
<td>2000</td>
<td>100.0</td>
<td>100.6</td>
<td>102.7</td>
<td>106.7</td>
<td>134.1</td>
<td>148.5</td>
<td>–</td>
<td>1862.0</td>
</tr>
</tbody>
</table>

## Lessons

Hybrid caching is sometimes a little better, **often much worse** – don’t use it

Caching can be very sensitive (almost chaotic)

60 experiments took a morning *in vitro*, a week *in vivo*
A Prototype Page Profiler

From Least to Most Frequent

“Frequent flyers” among 10,000,000 pages transmitted

<table>
<thead>
<tr>
<th>count</th>
<th>address</th>
<th>eip</th>
<th>client</th>
</tr>
</thead>
<tbody>
<tr>
<td>764210</td>
<td>08fbb6f0</td>
<td>086fe958</td>
<td>29</td>
</tr>
<tr>
<td>206058</td>
<td>0e6fc064</td>
<td>088c2b9</td>
<td>29</td>
</tr>
<tr>
<td>204235</td>
<td>085b1822</td>
<td>08081372</td>
<td>13</td>
</tr>
<tr>
<td>203168</td>
<td>084decc4</td>
<td>0809e388</td>
<td>13</td>
</tr>
<tr>
<td>181556</td>
<td>0e6fd114</td>
<td>088c5541</td>
<td>29</td>
</tr>
<tr>
<td>179481</td>
<td>0e232028</td>
<td>086ff492</td>
<td>29</td>
</tr>
</tbody>
</table>

...

Observations

About 7½% of the pages come from one instruction
Second place is only about 2%

Time to check that instruction

Add one new essential piece: Pointer to executing instruction
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 is causing 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 reduces the number of page faults and the amount of data transmitted by about 7.5%
On Performance Bugs

Definition: A performance bug is
a minor glitch
that does not alter the correctness of a program
but does cause it to consume excessive resources.

A Real Instance

Failure to turn off a unit test

Bugs Waiting to Bite

“This constant 10007 is weird. I’ll change it to a nice round
10000.” (Or 8000, or 8192, or …)

“A new algorithm gives a 10% speedup on this test.”
And a factor of 25 slowdown elsewhere

“It is faster in C to search a list from back to front.”
130 comparisons $\rightarrow$ (10000 – 130) = 9870 comparisons
A (Monster) Performance Bug

E-mail to the SWDup Team

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

```c
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 200\textsuperscript{th} 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

\[
\frac{N}{200} \rightarrow N - 200
\]

Doubles or triples page faults

Increases CPU utilization by about 50%
Data Compression Notes: Executive Summary

D made a set of 5000 pre-post pairs, each of length 4096.
The current diff represents those in about 338Kbytes.
Changing the cutoff from 64 to 512 reduces the space to 170Kb (a reduction to about 50% from one #define).
Changing from a long-wise to a char-wise diff further reduces the space to 126Kb (to about 37%).
Additional compression might be able to reduce that char-wise space further (to 90Kb, or 27%).
Summary: Optimal Cutoff Size?

Current cutoff size is 64

When to send complete page versus set of changes?

Each change is 6 bytes: 2 bytes of address, 4 bytes of long

Optimal cutoff when $6 \times \text{changes} = 4096$, or 683 changes

Experiments on D’s data

<table>
<thead>
<tr>
<th>Cutoff</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1087186</td>
</tr>
<tr>
<td>32</td>
<td>662234</td>
</tr>
<tr>
<td>64</td>
<td>337610</td>
</tr>
<tr>
<td>128</td>
<td>230456</td>
</tr>
<tr>
<td>256</td>
<td>178758</td>
</tr>
<tr>
<td>512</td>
<td>170492</td>
</tr>
<tr>
<td>1024</td>
<td>172530</td>
</tr>
<tr>
<td>2048</td>
<td>176626</td>
</tr>
</tbody>
</table>
Tragic Intrusion of Reality

R,  We changed the cutoff value and found that the bandwidth dropped from 1.9Mbps at 64 to 1.7Mbps at 512, for a reduction of 10.5%.

The reason for the small gain is a misinterpretation between R and D about the format of the files. Nevertheless, the penalty (performance and memory) for making the change is virtually nil so there’s little reason not to do it. – D

New experiment on good data: old reduction to 50.5% now becomes 59%

What else could be going on?

Experiments ignore the “dup link headers”– “in the noise” for 4KB pages, but substantial for 34-byte pages

This “essential data” missed the essence!
Packet Overhead

R, Here’s the 28 bytes of packet overhead. Some of it may be avoidable, and some not. – D Teddy Bear Reductions

4 Header (2 bytes for type, and 2 for length).
4 Version ID. Probably not really necessary.
1 Client ID. Combine 5 bits and 2 bits in one byte
1 Message type: full page, partial page or audit.
2 Number of differences or audits. Use header length
4 Checksum. Possibly unnecessary.
4 Transaction ID. No longer necessary, and I will eliminate this in the near future.
4 Address of the instruction that caused the page fault on this page. Only needed for debugging.
4 Address of the page, clearly necessary.

28 → 12 → 8 Lower 12 bits are 0, so only 2 ½ bytes
Summary of Data Compression

A Long and Winding Road
What is the real problem? Compression or diffing?

Incorporated into the System
Cutoff 64 → 512
Changes in packet overhead (28 → 12)
Simple fixes were “good enough”

Waiting in the Wings, Just in Case
Byte-level compression
Further changes to the packet

A Powerful Technique
Gather essential data in vivo for in vitro experiments
Iterate to get the true essence
Summary

Algorithms Used in the System

Cache: Kept pure LRU
Cache search: Doubly-linked list with hash function
Change cutoff and compress packet size
How much improvement?

Insights

Found patterns to investigate
  Squeeze pairs into one; remove long sequences
Identified performance bugs
Increased respect for sensitivity and performance bugs

Tools

Little *in vitro* experiments on essential data collected *in vivo*
Avaya Labs Cup Citation

The Software Duplication Team has overcome many barriers and has achieved results that most people thought were impossible: making the performance of software-based duplication better than that of hardware-based duplication.

The team also came up with many innovative ideas to improve performance and each member of the team was very proactive in resolving the many issues that were discovered while working on the solution.

This is a great result from great teamwork. Congratulations!
Lessons for Programmers

Useful Themes

Simplify!

*In vitro* experiments on *essential data* collected *in vivo*

Families of little programs – 50-100 lines of code

Dealing with Reality

Find the right problem

Work with real data

Keep your eyes open for interesting patterns

Persistence

Expect lots of wrong turns and dead ends

Always conduct more experiments

Ain’t no horse …
Essential Data

Key Parameters
   Collect *in vivo*, study *in vitro*

SW Dup
   Cache (address, client) pairs
   Cache pairs, plus executing instruction
   Complete cache pages (4K old/new pairs)

Other Examples
   TSP – Points in k-space (circuit board holes, precincts)
   1-d bin packing – Weights (file sizes)
   2-d bin packing – Rectangles (VLSI chips, paintings)
   Storage allocation – new/free request sequence
Lessons for People

Communication

Offer a menu of solutions
   Ideally from simple yet good enough to optimal if complex
Pay in the coin of the realm
   E-mail, PowerPoint, sample code, little simulators
Be the best consultant you can be
   Ask dumb questions; give credit where it is due

Try Your Hand at History

Invest in the Relationship
Some Long-Term R Investments

1980: Ravi Sethi’s University of Arizona class on Software Tools, taken by Boyle
   Rapid development

1985: Bentley and McGeoch, “Amortized analyses of self-organizing sequential search heuristics”, CACM
   Analysis of Move-To-Front

1985: Bentley, Cleveland and Sethi, Experiments on hash functions reported in the Dragon Book, 2nd Edition
   Experience with hash functions

1986: Bentley, Sleator, Tarjan and Wei, “A locally adaptive data compression scheme”, CACM
   Small experiments on compression

1996, Krishnan, “Online prediction algorithms for databases and operating systems”, Brown University Ph.D.
   Analysis of policies

1999: Bentley and McIlroy, “Data compression using long common strings”, Data Compression Conference
   Useful tools

One last story
   D: “Sure, I know Ravi. He’s my friend!”