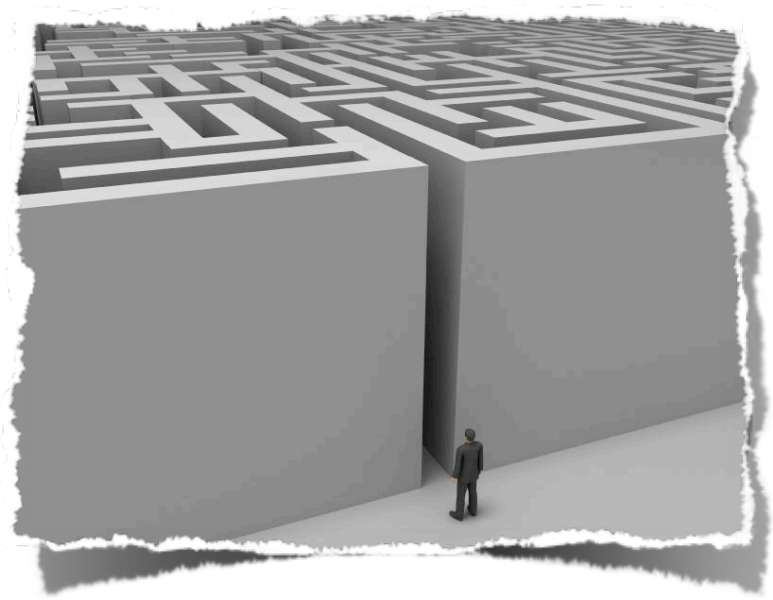
Mastering the Empirical Maze

Laurent Michel
University of Connecticut



Overview

- Motivation
- Empirical science
- Empirical method in CS
  - Specificities
  - Pitfalls
  - Platforms
- Analysis

Motivation

- A few motivations

Empirical Method is not that easy
Essential to do it right!

Computing makes it even *harder*

Current trend in paper is
scary, disappointing...
and it is getting worse!

Why? Oh Why?

- What is the purpose of an experiment?

1. Validate a scientific hypothesis

2. Convince other scientists of the validity

Keys

Clear
Sound
Reproducible

What *are* the problems?

- Clarity
  - Not enough information communicated
    - Setup, conditions, environment, ...
- Soundness
  - Validating on irrelevant aspects
  - Wrong measurements
  - Statistical significance
  - ....
- Reproducibility
  - Simply can't reproduce!

Overview

- Motivation
- Empirical science
- Empirical method in CS
  - Specificities
  - Pitfalls
  - Platforms
- Analysis

Empirical Sciences

- The scientific method

"a method of procedure that has characterized natural science since the 17th century, consisting in systematic observation, measurement, and experiment, and the formulation, testing, and modification of hypotheses."

Oxford English Dictionary

Empirical method

- Central tenet
  - All evidence for the scientific method must be
    - Empirical or Empirically based
    - Meaning....
      - Dependent on *observable* evidence

Physics Example

- Question
  - What kind of relation/force (if any) exist between masses?
- Observation
  - If I hold an apple....
  - And let go of it....
  - It falls!
- Hypothesis
  - Maybe the two masses are attracted to each other
  - So: Posit some relation and check what works!

Physics Example

- Newton's theory of universal gravity
  - 2nd law:
  - Force is proportional to the masses
  - Force is inversely proportional to the distance
  - Environment influence is a constant factor (no vacuum)

$$F = G\frac{m_1 \cdot m_2}{d^2}$$

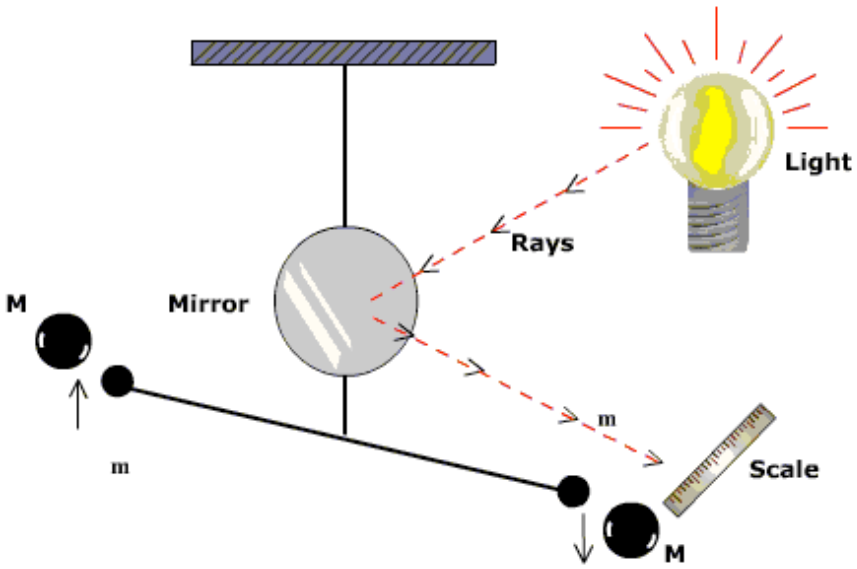
- It has
  - Limits (and therefore assumptions)
- It can be tested
  - With experiments that either
    - Confirm
    - Infirm

Cavendish's Experiment

- Simple Idea
  - Measure the amount of "torsion force" on wire when...
    - The large balls get close to small balls
    - All balls have known masses
    - Distances from center to center can be measured
  - Check that it satisfies

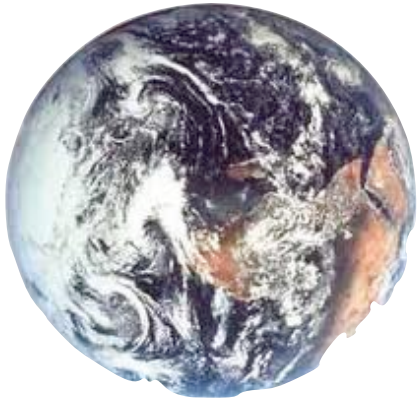
$$F = G\frac{m_1 \cdot m_2}{d^2}$$

  - Bonus
    - We determine constant G



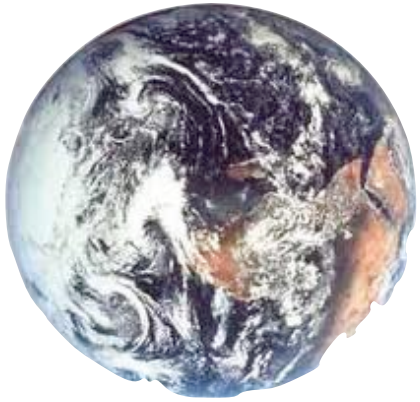
Empirical method

- Important facts
  - Apparatus exist in the **real world**
    - Observe that nature conforms to prediction



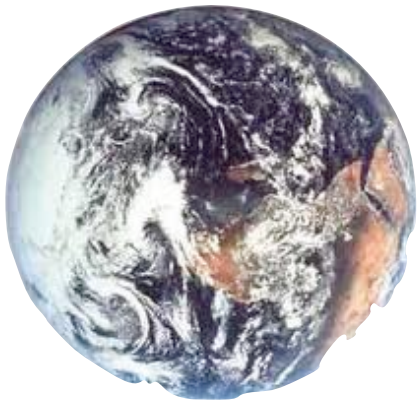
Empirical method

- Important facts
  - Apparatus exist in the **real world**
    - Observe that nature conforms to prediction
  - Measurements are imperfect
    - Inherent noise in mass, distance, torsion force



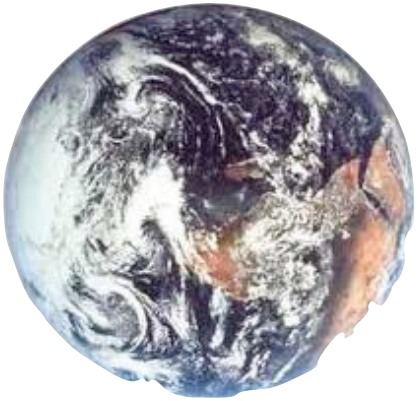
Empirical method

- Important facts
  - Apparatus exist in the **real world**
    - Observe that nature conforms to prediction
  - Measurements are imperfect
    - Inherent noise in mass, distance, torsion force
  - Environment matters
    - G changes with location [earth, moon, jupiter....]



Empirical Method must inherently handle this *variability*

Process



- The problem
  - First, formulate an hypothesis
  - Then, determine how to validate it
- Given an hypothesis...
  1. Formalize hypothesis / assumptions
  2. Design an experiment to see whether predictions are met
  3. Determine the *conditions* of the experiment
  4. Determine the *measurements* needed to validate
  5. Determine *how to deal with the uncertainty*
  6. Execute the experiment
  7. Analyze the outcomes

Question

How much of this carries over to CS?

Overview

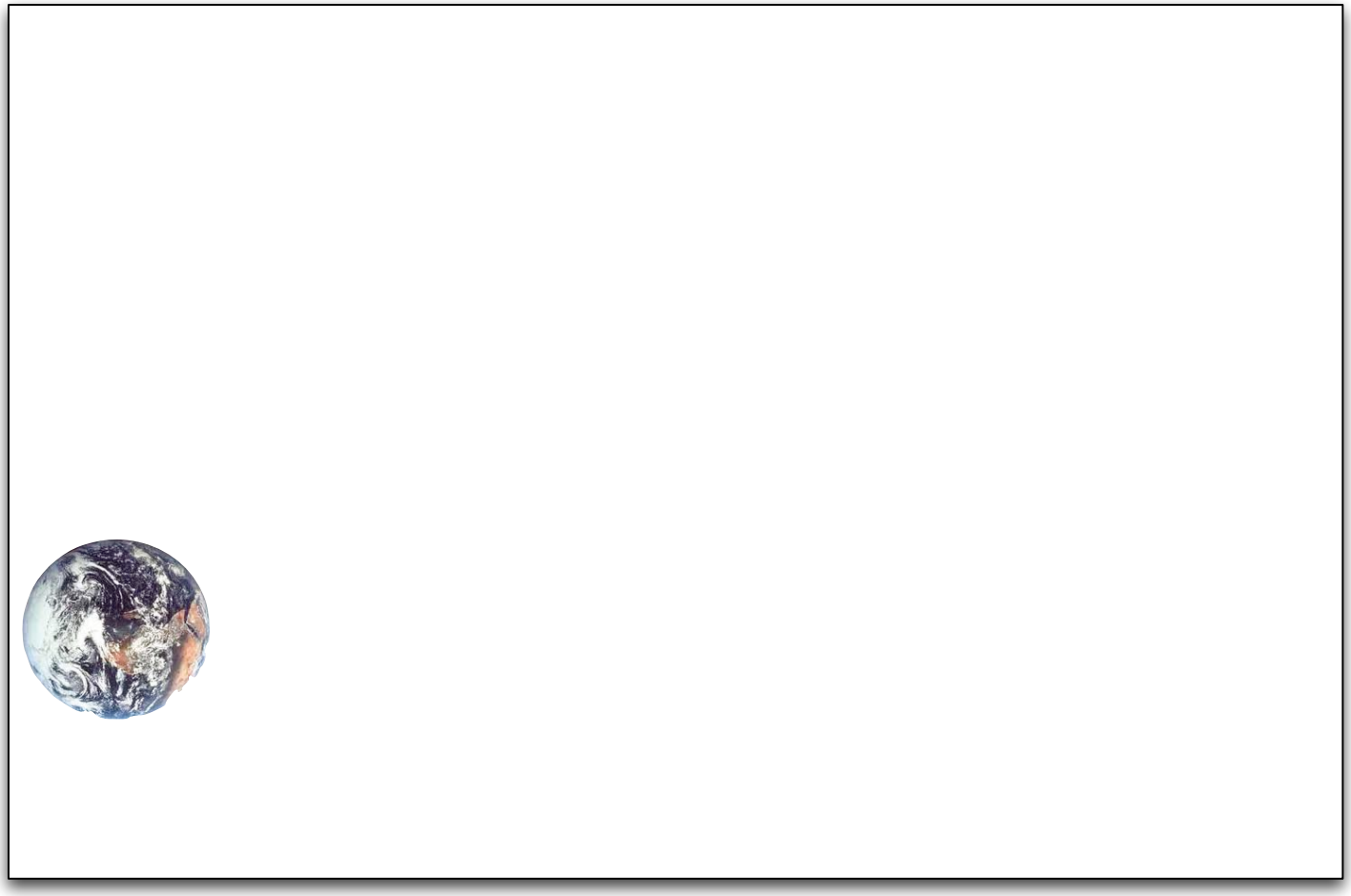
- Motivation
- Empirical science
- Empirical method in CS
  - Specificities
  - Pitfalls
  - Platforms
- Analysis

Empirical Method in CS

- It's a *different world*!

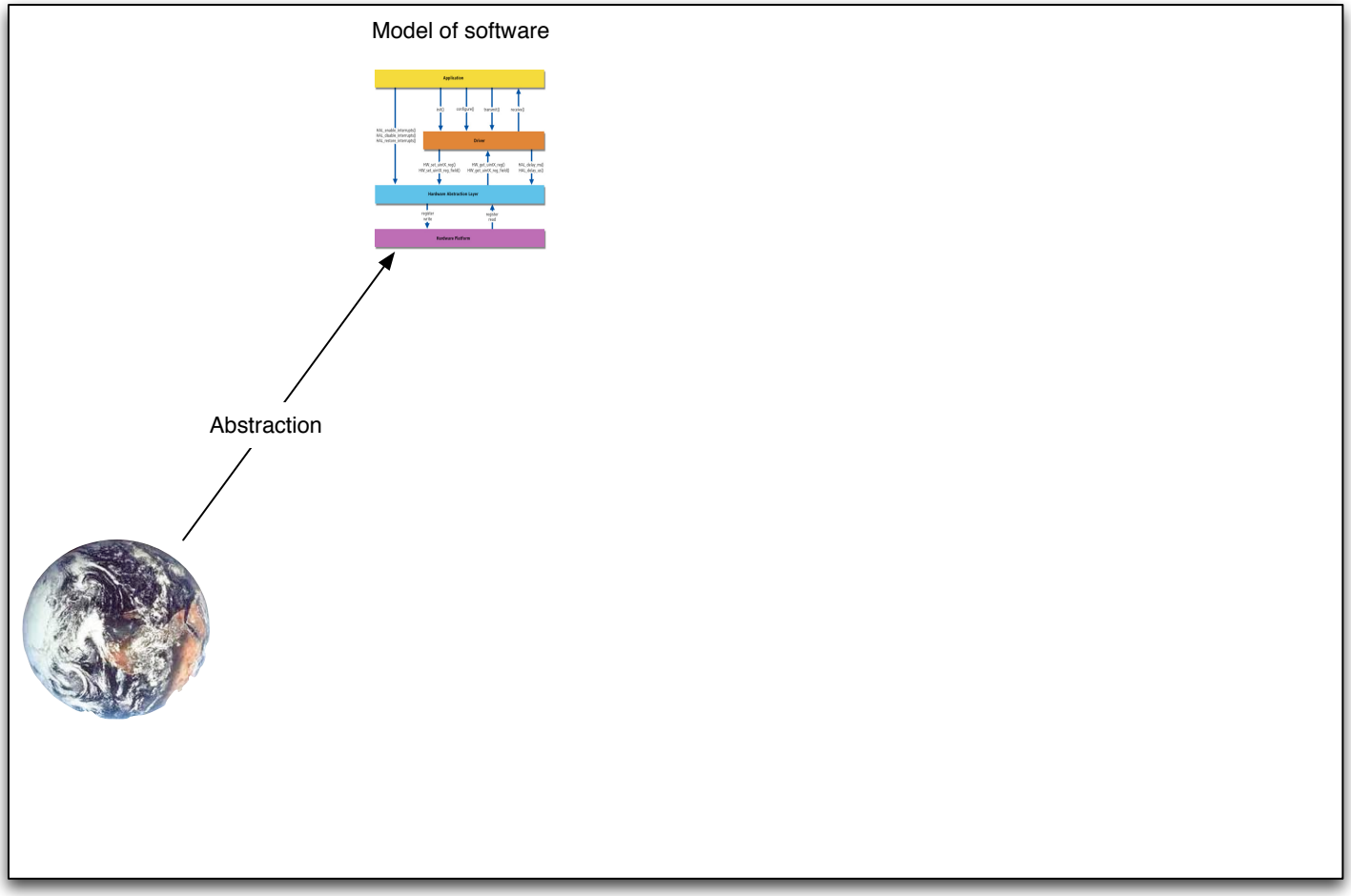
What we build

- Software as a *model of the real world*



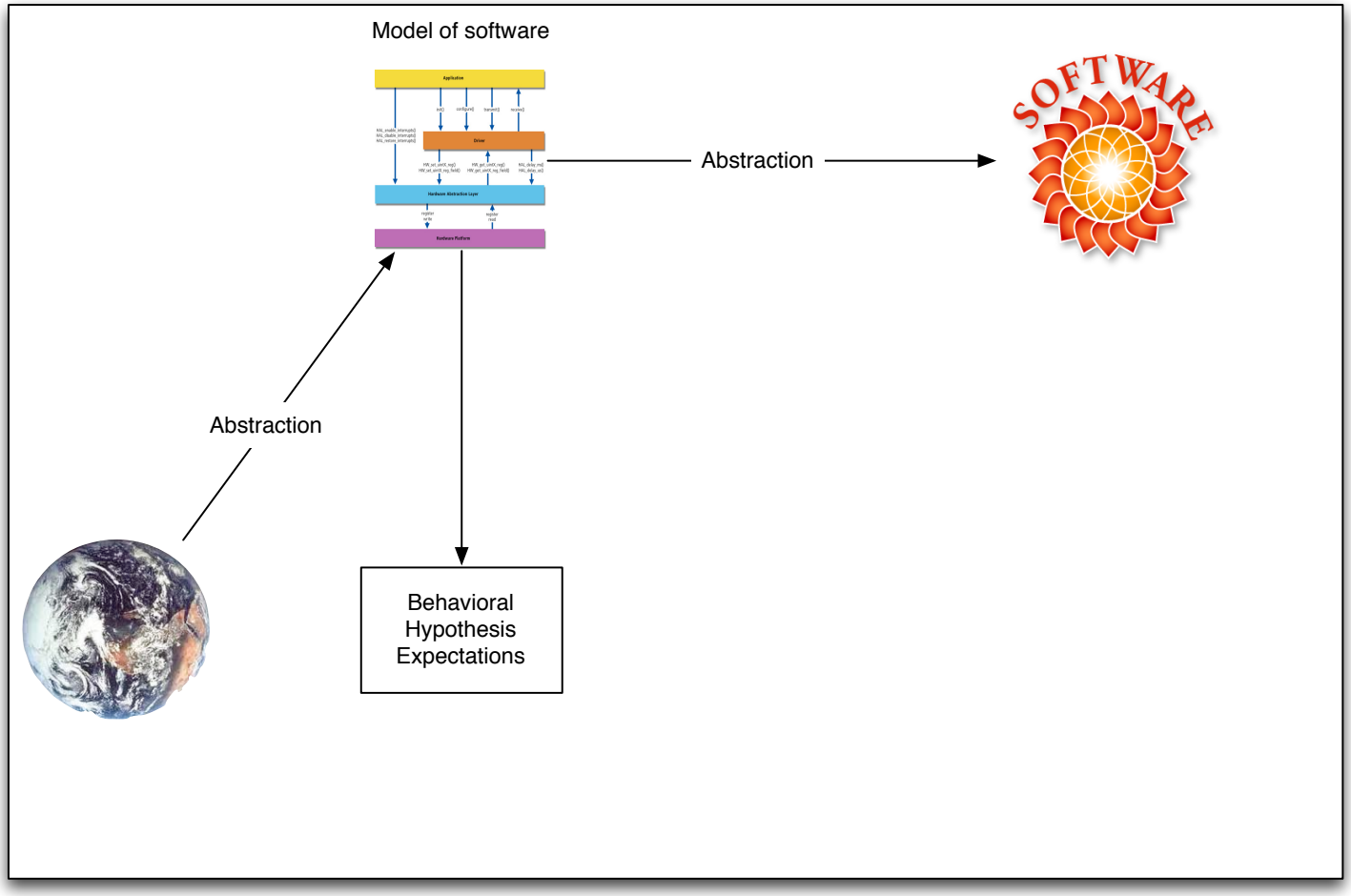
What we build

- Software as a *model of the real world*



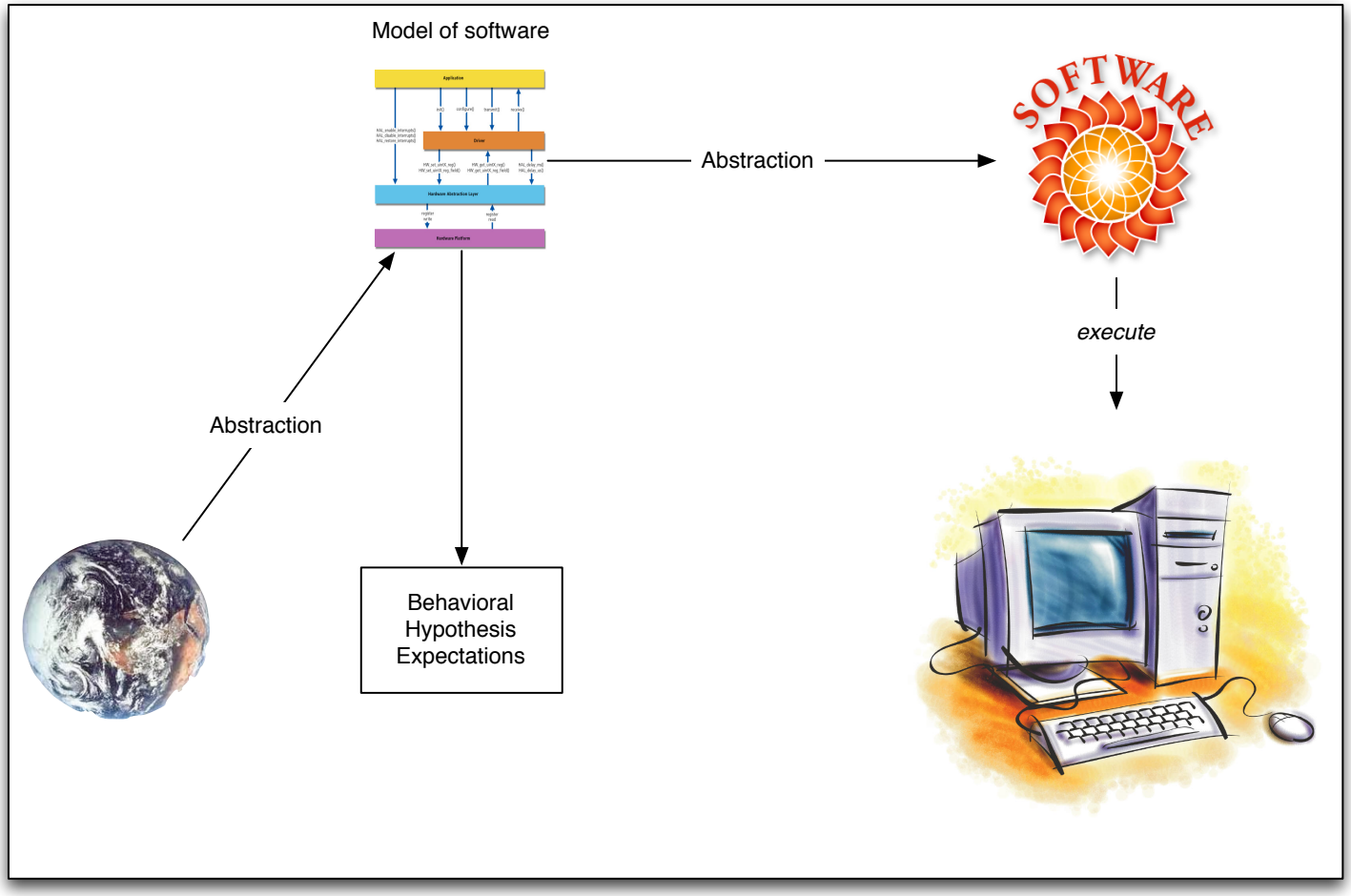
What we build

- Software as a *model of the real world*



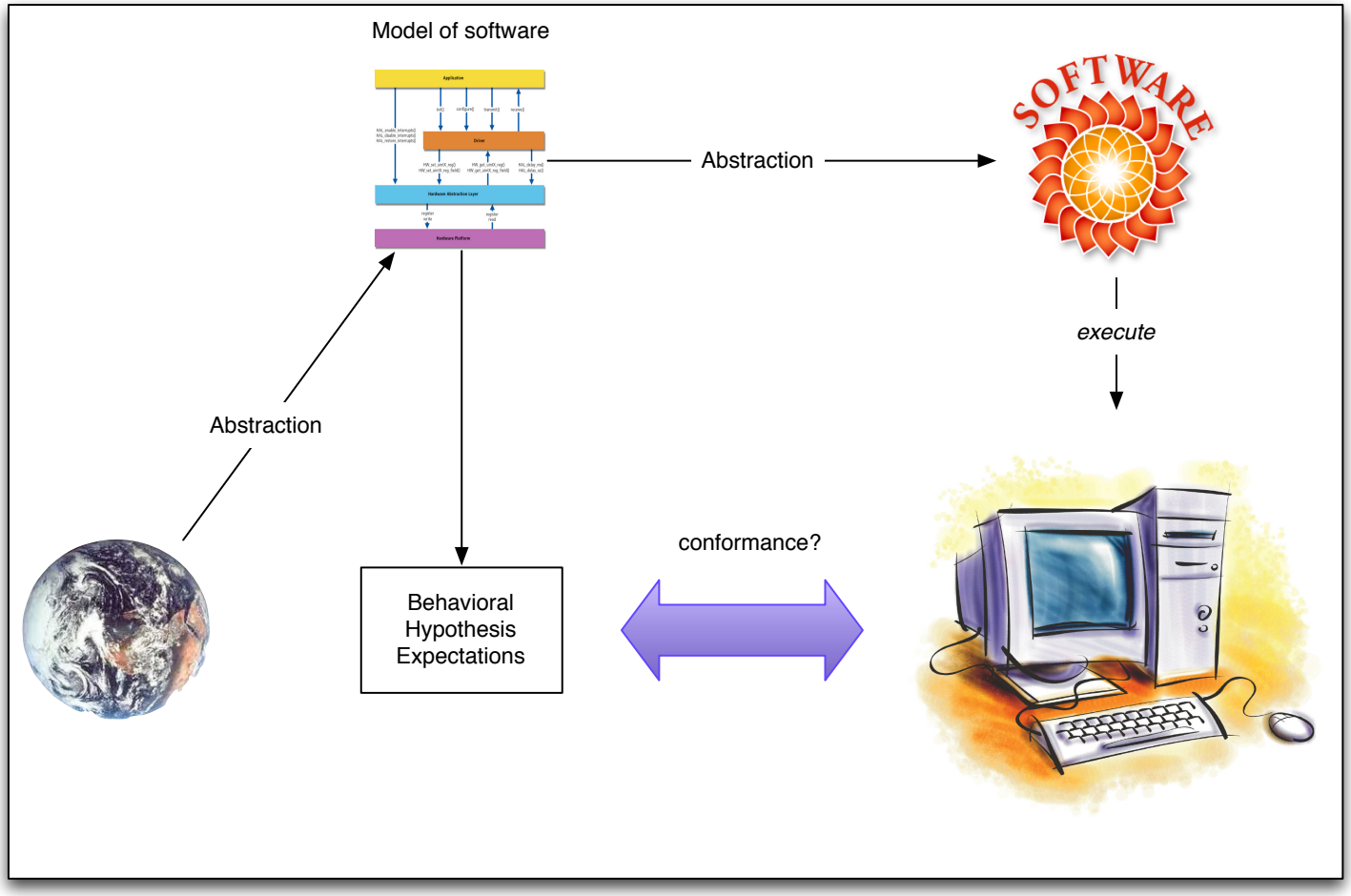
What we build

- Software as a *model of the real world*



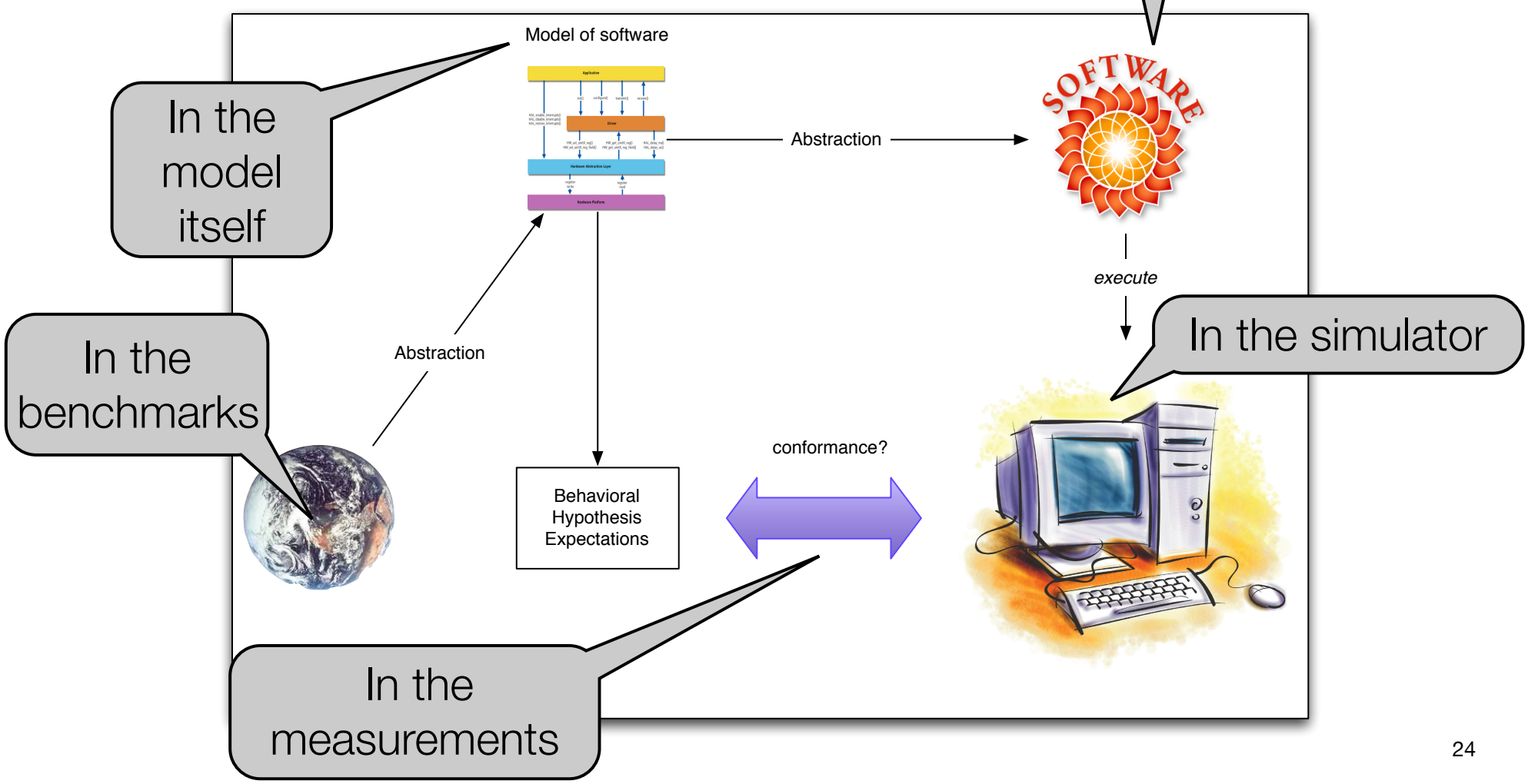
What we build

- Software as a *model of the real world*



Caveat Emptor

- Uncertainty / Stochasticity lurks...



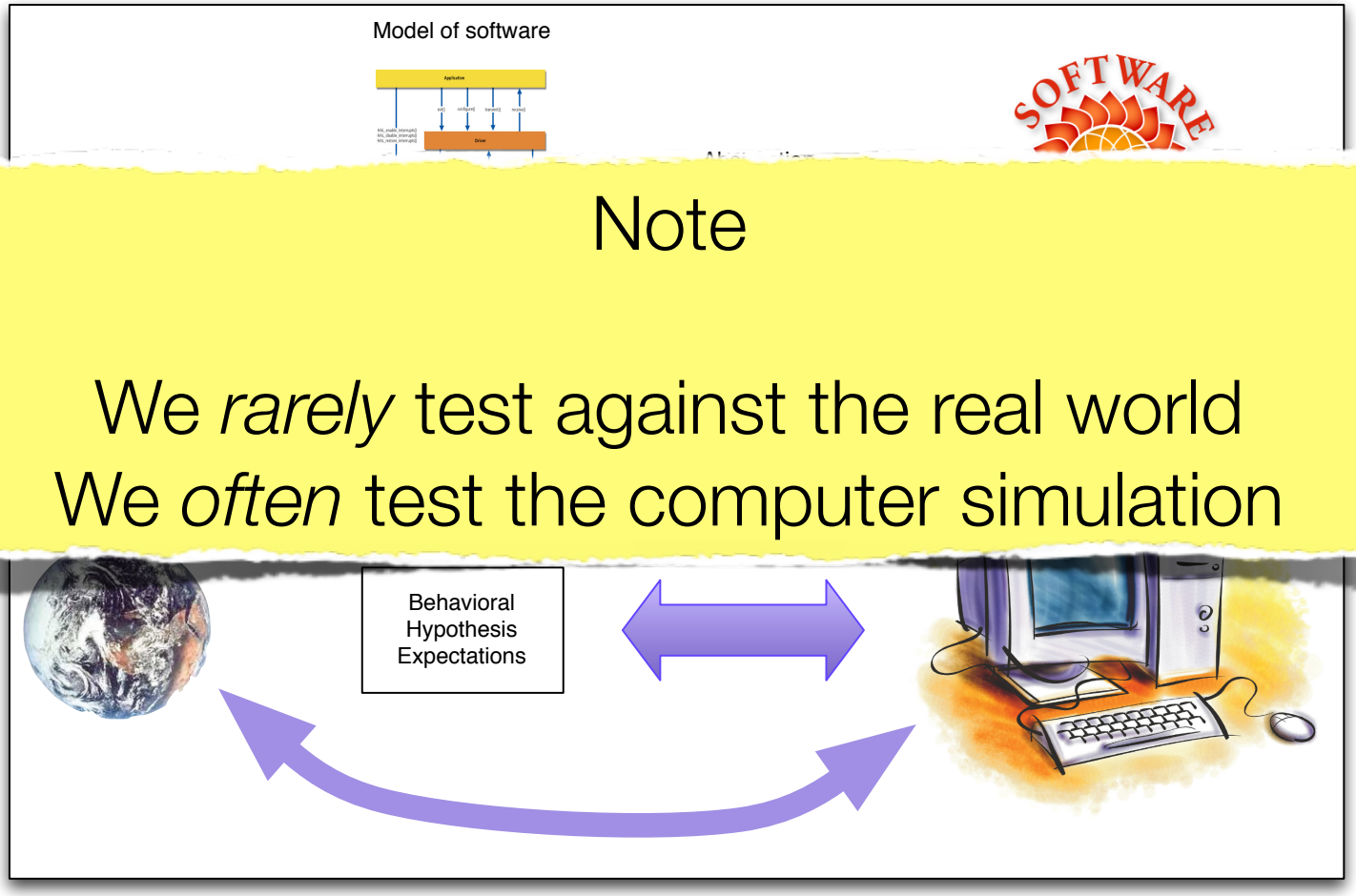
What can go wrong ?

First Recall

- We need an *experiment*
  - To **validate**/Test the hypothesis
  - That is **tractable** [doable in the allotted time]
  - For which we can make **pertinent** measurements
  - Where the measurement **uncertainty is minimal**
  - That is **reproducible**

First Pitfall

- What about this conformance?

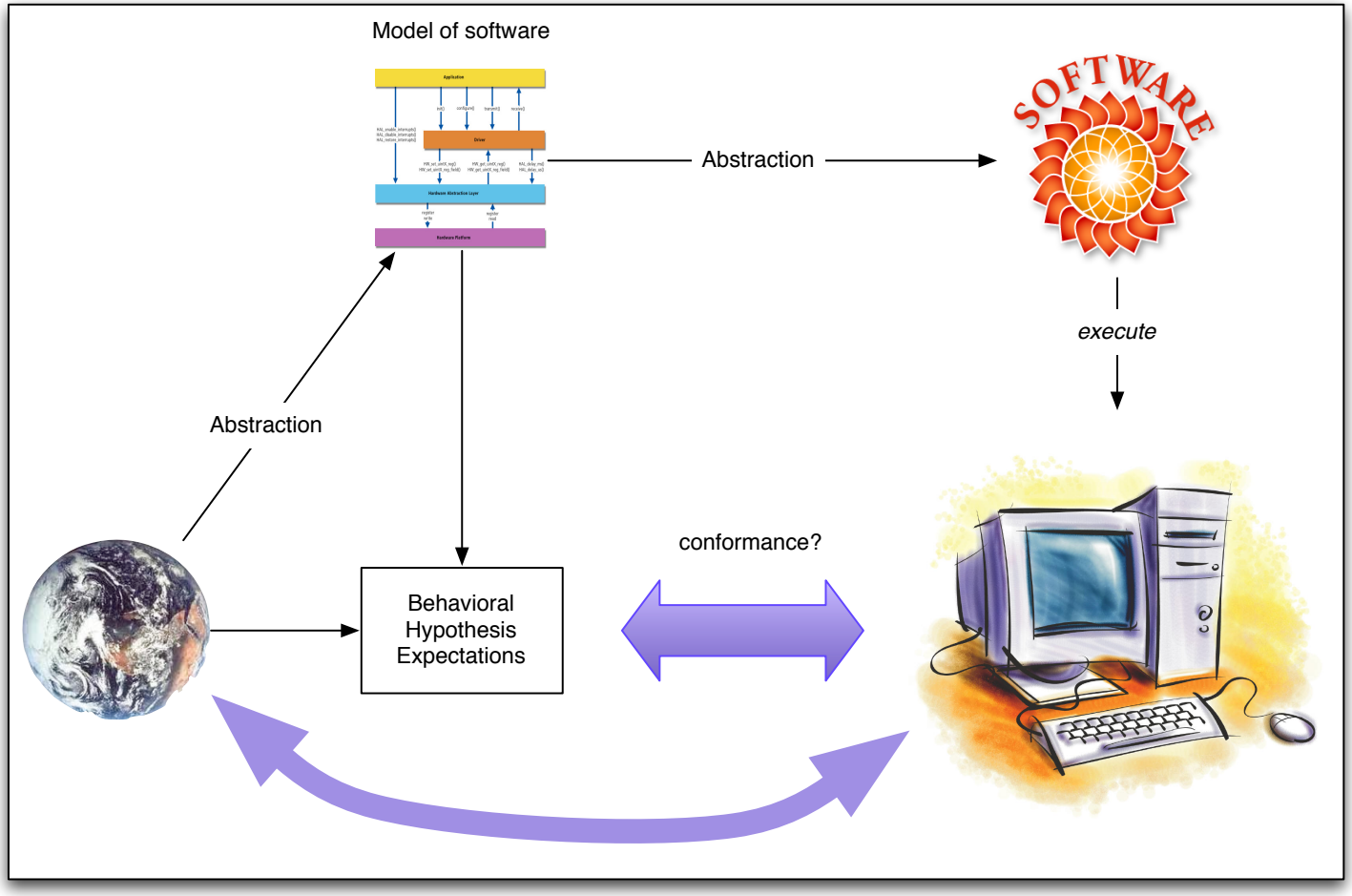


Bottom-line

- If the abstraction is flawed....
  - The simulation might conform to the hypothesis of a flawed model!
  - But the entire result is irrelevant in practice!
- The Pitfall

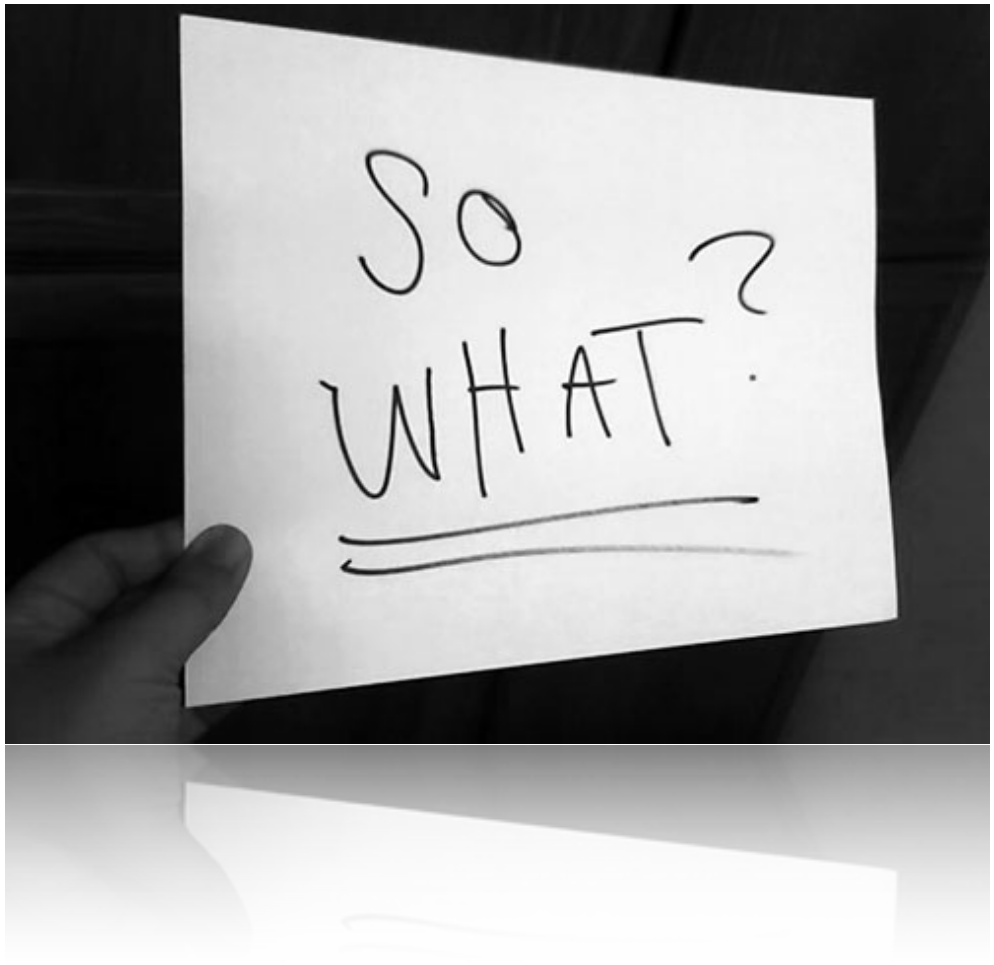
The behavioral expectations should
not derive from the model (alone)
but from the real world!

Refined model



Second Pitfall

- The experiment occurs *on the simulator*



Change

What has changed

Then	Now
dedicated	Time-sharing
slow with constant speed	*Very* fast, variable speed
Static speed	Power-driven resource allocation
flat memory hierarchy	NUMA
fixed CPU caches	variable cache allocation
small memory	Huge memory
Mono-processors	Multi-processors
Homogeneous	Heterogeneous
in-order sequential	deep pipelining speculative, out-of-order
Minimal OS	Complex OS, caching
Mono-task	Multi-task

Bottom-line

- Without enough details...
  - It is impossible to reproduce timing performance reliably
- Today, it depends on
  - *Exact* machine being used [Ghz myth anyone?]
  - Laptop or Desktop [why?]
  - Other tasks running?
  - Which OS? [and which version!]
  - Anything running concurrently? [are you sure?]
  - Which compiler did you use?
  - With what options?

The compiler? Really ?

- Yes! [http://www.luxrender.net/forum/viewtopic.php?f=21&t=603]
  - That was in 2008, the gap is certainly not shrinking....

Compiler	Options	Speed
gcc	-O2 -Wall -DLUX_USE_OPENGL -DHAVE_PTHREAD_H	**factor: 1.0**
gcc	**-O3 -march=prescott -mfpmath=sse -ftree-vectorize -funroll-loops** -Wall -DLUX_USE_OPENGL -DHAVE_PTHREAD_H	**factor: 1.1419 (+14.19%)**
gcc	-O3 -march=prescott -mfpmath=sse -ftree-vectorize -funroll-loops **-ffast-math** -Wall -DLUX_USE_OPENGL -DHAVE_PTHREAD_H	**factor: 1.1677 (+16.77%)**
gcc/profile	Pass 1 =>"-O3 --coverage -march=prescott -mfpmath=sse -ftree-vectorize -funroll-loops -ffast-math -Wall -DLUX_USE_OPENGL -DHAVE_PTHREAD_H" Pass2 => "-O3 -fbranch-probabilities -march=prescott -mfpmath=sse -ftree-vectorize -funroll-loops -ffast-math -Wall -DLUX_USE_OPENGL -DHAVE_PTHREAD_H"	**factor: 1.2117 (+21.17%)**
icc	Pass 1 =>"-prof-gen -prof-dir /tmp -O3 -ipo -mtune=core2 -xT -unroll -fp-model fast=2 -rcd -no-prec-div -DLUX_USE_OPENGL -DHAVE_PTHREAD_H '-D"__sync_fetch_and_add(ptr,addend)=_InterlockedExchangeAdd(const_cast<void*>(reinterpret_cast<volatile void*>(ptr)), addend)"'" Pass2 => "-prof-use -prof-dir /tmp -O3 -ipo -mtune=core2 -xT -unroll -fp-model fast=2 -rcd -no-prec-div -DLUX_USE_OPENGL -DHAVE_PTHREAD_H '-D"__sync_fetch_and_add(ptr,addend)=_InterlockedExchangeAdd(const_cast<void*>(reinterpret_cast<volatile void*>(ptr)), addend)"'"	**factor: 1.4245 (+42.45%)**

Third Pitfall: Benchmark Selection Hell

- The Problem
  - How to build a benchmark suite?
- Namely
  - Use *relevant* benchmarks [most likely to trigger the behaviors]
  - Use the *right-size* benchmarks
  - Precisely specified [for reproducibility]
  - Measurements cannot be intrusive
  - Broadly available [for reproducibility]

Relevance I

- *Choice is delicate*
- Some benchmarks are tuned for a specific technique!
- Example
  - Un-capacitated facility location
  - ORLIB source: [http://people.brunel.ac.uk/~mastjjb/jeb/orlib/capinfo.html]
    - *Small scale.*
    - *Hard for MIP methods*
  - Kratica source [Solving The Simple Plant Location Problem By Genetic Algorithm]
    - *Much bigger!*
    - *Designed with specific structures*

Relevance II

- Critical
  - If the benchmark does not exert the piece to measure...
  - It becomes useless!
- Example
  - Imagine testing a new way to implement backtracking search
- Quizz
  - What is the hypothesis ?
  - What kind of benchmark should be chosen to evaluate?

Use case: The hypothesis

The new implementation is faster than traditional implementations of DFS in modern solvers.

Use case: Benchmark selection

- Choose a “pure” approach
  - Benchmarks that
    - Propagate very little [as little as possible]
    - Backtrack as much as possible!
  - This is the worst case scenario.
  - Pitch the benchmark *against* you.
- Therefore
  - If you win here, you will win under better circumstances
  - The results are independent of the propagation used

A New Constraint

- Paper topic
  - A new constraint / propagator
- Issues
  - Does it achieve the same filtering?
  - Does it claim better complexity?

Filtering issue

- If the filtering is different...
  - The constraint may induce changes in dynamic heuristics
  - The constraint may induce different filtering at each fixpoint
- How to evaluate?
  - Separate the effects [evaluate with static branching]
  - Measure filtering volume
  - Consider micro-benchmark on constraint alone
  - Consider macro-benchmark effect [in context!]

Complexity issue

- Question
  - Is the improvement relevant in practice?
- Traps
  - Depends on time spent in *that* propagator
  - Time spent in propagator depends on benchmark too!
  - Improvement might not be significant on problem

Right-sized

- If benchmark is too big
  - You can't complete all the test in due time
- If benchmark is too small
  - You might well fall under your ***measurement noise floor***.
  - Hence, any measurement is pure noise and useless.

Precisely Specified

- Meaning
  - The data set is not sufficient.
- Experiment in CP often needs
  - A detailed model
    - Constraint
    - Search
  - A data set

Example

- Consider the statement

> "We used the langford 3/9 instance to test the search procedure XYZ." [anonymous]

- What is missing?

Example

- Consider the statement

> "We used the langford 3/9 instance to test the search procedure XYZ." [anonymous]

- What is missing
  - Which model was used?

Example

- Consider the statement

> "We used the langford 3/9 instance to test the search procedure XYZ." [anonymous]

- What is missing
  - Which model was used?
  - Given a model, which filtering algorithms were used for each constraint?

Example

- Consider the statement

> "We used the langford 3/9 instance to test the search procedure XYZ." [anonymous]

- What is missing
  - Which model was used?
  - Given a model, which filtering algorithms were used for each constraint?
  - Since XYZ uses randomization, what were the tie breaks?

Bottom line

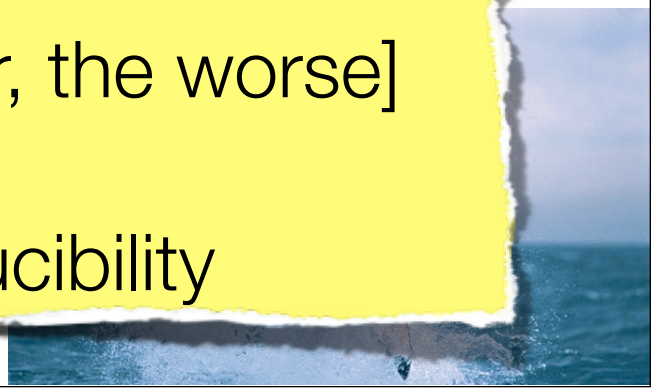
- Benchmark choice matters a lot
  - Can frustrate people trying to reproduce/understand
  - Can lead you astray
  - Can prompt you to draw incorrect conclusions
  - Size matters
    - Too big/too many, and you can drawn
    - Too small/too few, and you can miss the mark

Keys

Beware of stochasticity [the smaller, the worse]

Don't be fooled by large sizes

Be deliberate and strive for reproducibility



Fourth Pitfall: Tie break

- Tie-breaking
  - Or how to pick from several, apparently equally good, choices
- Can be done in two ways
  - Deterministically
  - Randomly

Deterministic tie-breaking

- Simple
  - Form a lexicographic ordering instead
- For instance
  - When domain sizes are equal, always prefer the *first variable*
- Issue
  - What determines who is first?
- Typical answer
  - Internal variable identifier
  - Depends on order of creation of variables
- Side-issue
  - Modeling object like matrices make it harder [row major?]

Randomized tie breaking

- Key idea
  - From the set of equivalent variable
  - Draw one uniformly at random.
- Issues
  - How do you deal with several invocation of the tie-break?
  - How do you deal with multiple tie-break sites?
  - How do you deal with multiple runs of the algorithm?

Of random number generators

- Notoriously delicate
  - They are deterministic at heart
  - Based on congruence relation
  - Require 64-bit wide arithmetic to get 32-bit wide pseudo-random
  - Some OS/Platform have *extremely bad* random generators....

Bottom-line

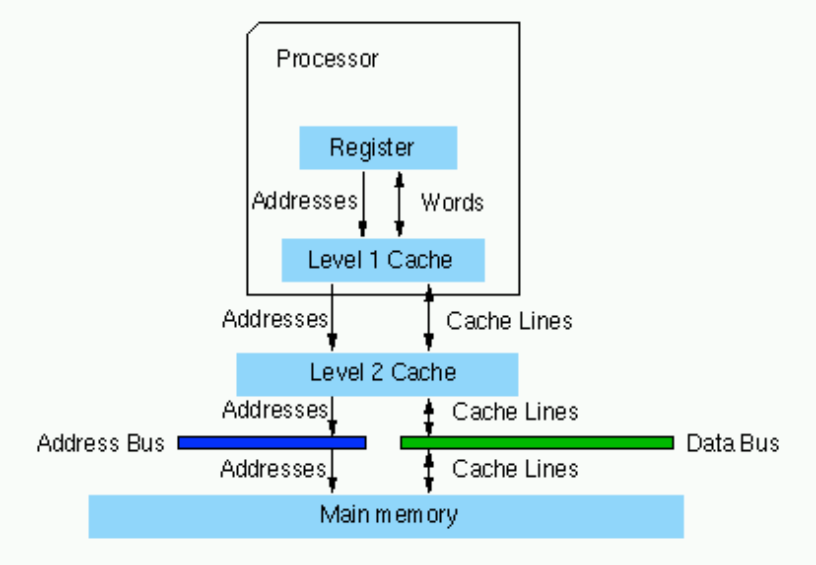
- Key insight
  - We can have multiple independent streams
  - But we must maintain the seeds for each stream
- It addresses the issues related to
  - Multiple invocations
  - Multiple sites
- For multiple runs
  - You must randomize the seeds too!
- Fundamentally
  - You ought to specify what you use

Overview

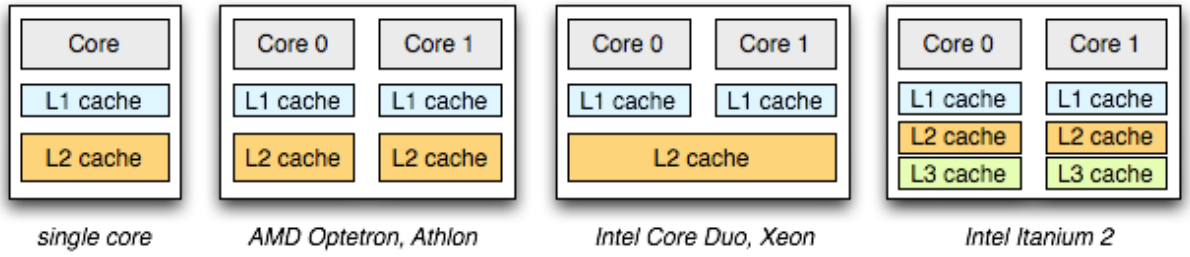
- Motivation
- Empirical science
- Empirical method in CS
  - Specificities
  - Pitfalls
  - Platforms
- Analysis

Performance factor

- On modern hardware, what is the driving force?



Memory hierarchy



single core *AMD Optetron, Athlon* *Intel Core Duo, Xeon* *Intel Itanium 2*

Putting things in perspective

- Use MUL instead of SHIFT
  - 5 cycles
- Conditional branch mis-prediction
  - 10 cycles
- Cache miss to main RAM
  - 200-250 cycles

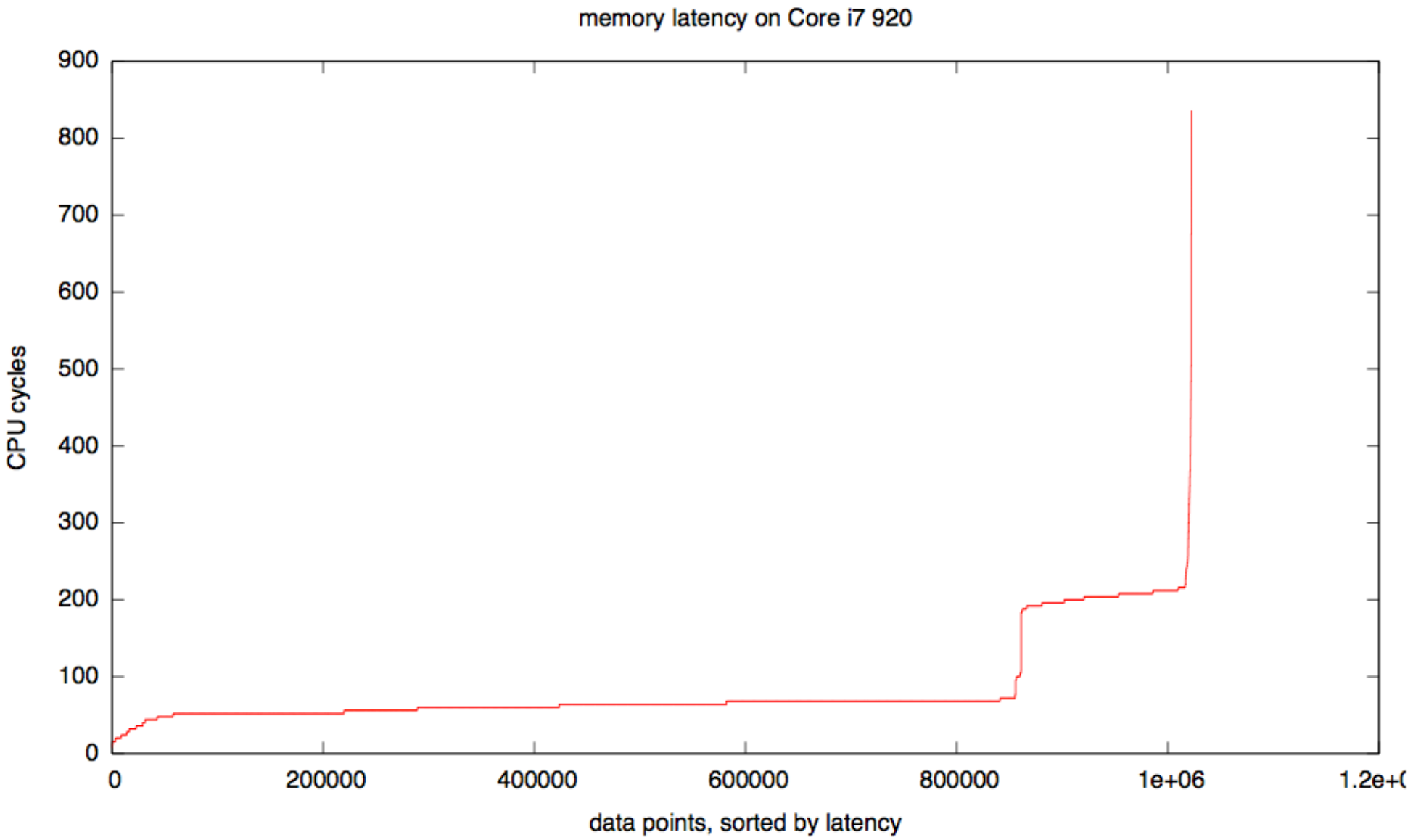
Putting things in perspective

- Memory access time (Linux running i7 920)
  - [http://www.linux-kongress.org/2009/slides/compiler%5Fsurvey%5Ffelix%5Fvon%5Fleitner.pdf]

Page Fault - IDE Disk	1.000.000.000 cycles
Page Fault - Buffer cache	10.000 cycles
Page Fault - RAM Disk	5.000 cycles
Main memory	~ 200 cycles
L3 cache	52 cycles
L1 cache	2 cycles

The Core i7 can issue 4 instructions per cycle. So a penalty of 2 cycles for L1 memory access means a missed opportunity for 7 instructions.

Non-uniform Memory Access Time [NUMA]



Bottom Line

- Your cache usage can have a dramatic impact on performance
- Do not run anything else on a cache sharing processor
- Possibly optimize your code to have better spatial locality

Measurements

- What should be measured ?
  - Whatever it is that can confirm/refute the hypotheses.
- Examples
  - Time performance [user vs. system time]
  - Space usage [VM usage vs. resident]
  - # of choice points [solver specific]
  - # of failures [solver specific]
  - Restarts [Strategy, diversification, learning]
  - Processors load [Parallel code]
  - Incrementality

Measuring time

- A delicate exercise
  - Modern processors (cores) share their cache
  - Modern processors have dynamic clock scaling
- Bottom line
  - Important that nothing goes on at the same time on the machine
    - No browsing, emailing, or listening to music. [why?]
  - On a dual core:
    - Either keep both core busy at all times [with same workload]
    - Or use only one core at all times

Measuring time

- A delicate exercise in its own right
- Many options
  - Use the OS-level time command

```
$ time comet benchcp/jobshop.co
...
real    0m7.485s
user    0m6.172s
sys     0m0.086s
```

  - This captures the entire runtime
    - real time [wall clock time]
    - user time [sum of time spent in user-land for all threads]
    - system time [time spent in system call on behalf of process]

Measuring time

- A delicate exercise in its own right
- Many options
  - Use system calls from your source

```
int t0 = System.getCPUTime();
...
int t1 = System.getCPUTime();
cout << “Elapsed CPU Time (user): “ << t1 – t0 << endl;
```

  - Finer grained instrumentation
    - Capture specific sections of the code
  - Caveats
    - Resolution of time
    - Cross-platform issues

Measuring on Windows

- sintx is a platform dependent [32/64] signed integer

```
SYSTEMTIME getSTARTTime() {
   FILETIME current;
   SYSTEMTIME now;
   GetSystemTimeAsFileTime(&current);
   FileTimeToSystemTime(&current,&now);
   now.wHour = now.wMinute = now.wSecond = now.wMilliseconds = 0;
   return now;
}
static SYSTEMTIME __onStart = getSTARTTime();
static int monthLength[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
sintx daysFromStart(SYSTEMTIME& now) {
  ...
}
sintx getCPUTIME()
{
    HANDLE me = GetCurrentProcess();
    FILETIME createTime, exitTime, kernTime, userTime;
    SYSTEMTIME now;
    int ok = GetProcessTimes(me,&createTime,&exitTime,&kernTime,&userTime);
    FileTimeToSystemTime(&userTime,&now);
    sintx elDays = daysFromStart(now);
    sintx retVal = now.wSecond * 1000;
    retVal += now.wMinute * 60 * 1000;
    retVal += now.wHour * 60 * 60 * 1000;
    retVal += elDays * 60 * 60 * 24 * 1000;
    return retVal+now.wMilliseconds;
}
```

Measuring on UNIX

- sintx is a platform dependent [32/64] signed integer

```
sintx getCPUTIME()
{
   struct rusage urusage;
   struct timeval utimeval;
   getrusage(RUSAGE_SELF,&urusage);
   utimeval = urusage.ru_utime;
   return 1000 * utimeval.tv_sec + utimeval.tv_usec/1000;
}
```

  - Resolution is milliseconds

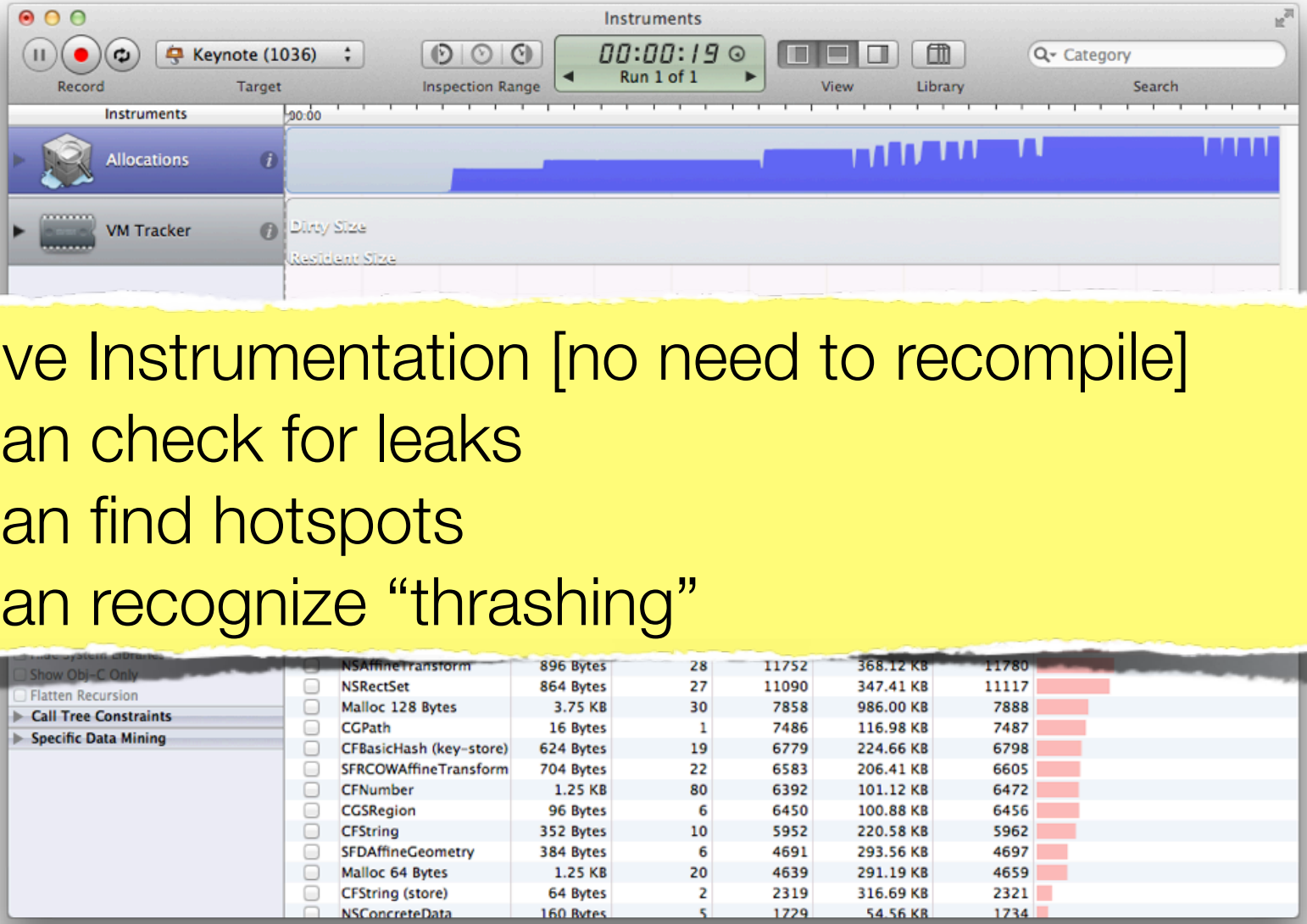
Very low-level measurements

- Use the builtin cycle counter of the CPU
- There are libraries for this!
  - http://www.ecrypt.eu.org/ebats/cpucycles.html
- Advantages
  - Very precise
  - Useful to measure effect of **low-level optimizations**
  - Measure each core/thread independently
- Limitations
  - 32-bit counter [it overflows regularly]
  - Doesn't stop while interrupts/system calls are taking place

Measuring Space

- Easier
  - At the OS level (MacOS), gain access to virtual memory usage
    - ps aux [global picture]
    - vm_stat [global picture]
    - vmmap [process picture. Entire address space detail]
  - At the process level (Again MacOS example)
    - Many tools to inspect usage.
    - Instruments
      - Leaks, malloc usage, VM usage,....

Instruments



Live Instrumentation [no need to recompile]
Can check for leaks
Can find hotspots
Can recognize “thrashing”

Measuring space

- Personally...
  - I have my own memory allocator
    - Finer-grained control over allocation algorithm
    - Instrumentation for space usage statistics
    - Debugging support (boundary guards)
- Alternatives (for debugging/instrumentation)
  - dmalloc http://dmalloc.com/
  - valgrind http://valgrind.org/

Measuring # choice points

- Be careful with this one!
  - Solvers count and report #choices differently
  - So the numbers are often not comparable across solvers!
- When comparing all the variants on the same solver...
  - It is fine.
  - It gives a sense of the size of the explored search space
  - For the search speed, consider #choices / second
- Above all
  - Don’t compare apple & oranges!

Measuring # of Failures

- A little better than # choices
- But still
  - Counting can vary with search
  - Counting can vary with what is considered a failure

```
using {
   forall(i in S : !x[i].bound()) by (x[i].getSize())
      tryall<m>(v in x[i].getMin()..x[i].getMax() : x[i].memberOf(v))
         m.label(x[i],v);
      onFailure
        m.diff(x[i],v);
   }
}
```

How is this counted?

Measuring parallel search

- This is a snake pit
- DO NOT
  - Measure parallel code with 1 thread vs. k threads
  - Use user-time to make the measurements
  - Assume that results will scale (even between known observations!)
  - Confuse parallel speedup with artifacts from parallel exploration
  - Compare to a *slow* sequential algorithm
    - Parallelizing slow code is *easy.*

Parallel search

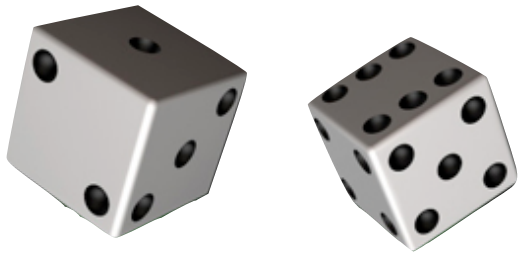
- Distributed computing point of view
  - "The best one can hope for is a linear speedup."
- Why?
  - The amount of work is known ahead and simply divided up
  - The parallel is not "smarter" than the sequential
- Why not?
  - Because we are solving COP! Better bound == more bounding!
  - Because we are using learning algorithms in search
    - Sharing of learned information => more effective search
  - Because we rely on tree search => we can get lucky! [on 1st sol]

Bottom line

- You must be extra-careful in what you measure
  - Especially for CSP/COP
- For instance
  - If the benefit are attributed to better pruning....
  - Then a sequential search that “jumps” in the tree would do well!
  - The speedup observation is
    - Not caused by parallelism
    - But caused by a “better” search!
- Instead
  - Measure performance on optimality proof!
  - Measure the amount of work as well.

Stochasticity

- Inherent and omnipresent
  - In the benchmark instances [online optimization]
  - In the models
  - In the benchmark families
  - In the measurements
  - In the simulator



Dealing with Stochasticity

- In the instance
  - An entire different line of optimization techniques [out-of-scope]

Dealing with Stochasticity

- In the model
  - Caused by tie-breaks
  - Caused by deliberate randomization
  - Caused by restarting [based on stability]
- Idea
  - Isolate each stochastic source
  - Many runs with different seeds
  - If possible, evaluate each source in isolation

Dealing with Stochasticity

- In the benchmark family
  - Artificial or real.
  - Useful to demonstrate *robustness*
- The objective
  - Show that the model works well across all instances in a class
  - Show that the model works well across several classes
- Pitfalls
  - Not all instances are equally hard [phase transition business]

Evaluate all instances of a class thoroughly
- to separate model induced stochasticity
- from intra-class stochasticity

Dealing with Stochasticity

- In the measurements
  - Runs that are too short may be below the timer resolution
  - That depends on the timer of course
  - That is affected by parallel code
- Idea
  - Do not run on “toy”/ “small” instances.
  - Run multiple times to average out these effects [with same seed]

Dealing with Stochasticity

- In the simulator
  - Uncertainty in measurements induced by
    - Cache behaviors
    - CPU frequency scaling
    - Artifacts from better bounds
    - Artifacts from better learning
- Idea
  - Run on a dedicated server
  - Don’t share caches. Always run in the same conditions/
  - Validate results (# of choices/# of failures should not vary)

Overview

- Motivation
- Empirical science
- Empirical method in CS
  - Specificities
  - Pitfalls
  - Platforms
- Analysis

Data Analysis

Lies, damned lies, and statistics.

Benjamin Disraeli (1804–1881)
Mark Twain (1906)

With lots of data...

- One must rely on statistics
  - To gain insights in the large volume of data
  - To compress the volume of information without loosing the keys
  - To better communicate with peers.
- A couple of simple observations....

Aggregation

- When lots of different benchmarks are used
  - It is tempting to aggregate the result and give a single runtime
- This is *less than ideal*
  - It **is much harder** to reproduce
  - It sheds no insights into the algorithms
  - Some benchmarks may completely dominate the totals
  - Averages are absolutely meaningless
  - Averaging the individual standard deviation is just as bad
- The only “ok” thing to do
  - Report the sum of the running times
  - Report the total number of time outs
  - But that is very coarse!

For performance measures

- Compute
  - Mean
  - Standard deviation
  - min / max / range
  - Empirical distribution [histogram]
- Advantage
  - It captures far more information about the population of runs
  - It captures information about robustness
  - It is not any harder to do!
- Don’t forget: at least 50 runs

Presenting the Data

- Tables are nice
- But graphics is often better.
- There are excellent tools for this.
  - Most notably: **The R Project** http://www.r-project.org/

The R tool

- R is a language and environment for
  - Statistical computing and
  - Graphics
- Huge amount of tools and material
  - Statistical tests [significance, conformance]
  - Time-series analysis
  - Classification / Clustering
  - Regressions
  - Tons of drawing/plotting facility [line,plot,chart,box,heat,....]
  - Produce nice PDF/PNG for inclusion in papers/talks
  - Reads data from CSV, DBMS (SQL)

One Example

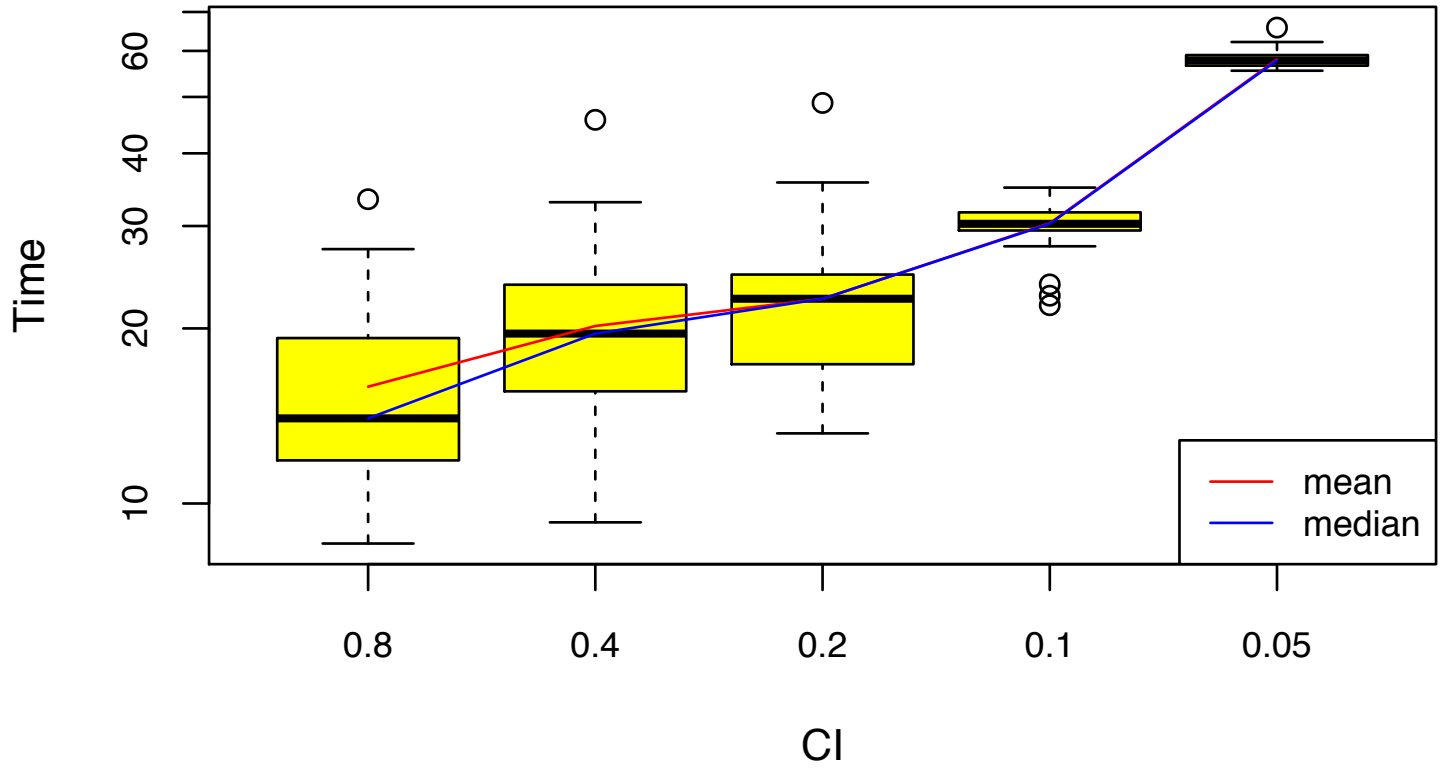
- Performance of ABS
- Objective
  - Measure the effect of the confidence interval parameter on the search
- Method
  - Fix all the parameters
  - Vary the CI parameter from 0.8 (loose) to 0.05 (strict)
    - [0.8,0.4,0.2,0.1,0.05]
  - Do 50 runs for each value

Raw data

CI,Run,C,F,I,T
0.8,0,30096,23126,406,26687
0.8,1,11987,8753,379,9037
0.8,2,14188,11351,412,13031
0.8,3,16744,12118,411,13379
0.8,4,75736,71120,443,27374
0.8,5,13352,9903,446,10589
0.8,6,62892,57266,408,21004
0.8,7,22590,16981,430,19250
0.8,8,66794,62422,446,21511
0.8,9,21865,16844,423,20538
0.8,10,8373,6919,404,8544
0.8,11,20901,15343,407,15412
0.8,12,16879,11469,439,11398
0.8,13,10845,8450,404,9095
0.8,14,74300,68034,430,26558
0.8,15,14053,10759,419,11507
0.8,16,17773,12979,419,11863
0.8,17,19754,15666,411,19191
0.8,18,18970,14008,409,15914
0.8,19,41105,36544,411,14298
0.8,20,22374,16362,420,18349
0.8,21,82887,77793,411,33361
0.8,22,13761,9971,407,11892
0.8,23,17800,13232,383,13903
0.8,24,17577,12390,389,12207
0.8,25,12214,8641,393,8681
0.8,26,20589,16106,363,18672
0.8,27,14155,11041,383,11676
0.8,28,18598,12930,410,14257
0.8,29,18965,14427,394,18360
0.8,30,69919,66122,372,21179
0.8,31,25475,20274,424,23633
0.8,32,13912,10992,387,13630
0.8,33,22118,16726,403,19789
0.8,34,14689,11341,400,13757
0.8,35,14052,11173,420,13878
0.8,36,14761,11390,435,12949
0.8,37,20831,14613,418,14508
0.8,38,8472,6974,415,8535
0.8,39,13001,9752,434,12070
0.8,40,18364,13813,376,13484
0.8,41,10547,8636,422,10298
0.8,42,24943,19208,429,23305
0.8,43,23276,16987,408,18071
0.8,44,12760,9285,440,10079
0.8,45,68410,62685,424,23235
0.8,46,17318,12669,416,13654
0.8,47,12610,9753,414,11244
0.8,48,20689,15974,413,15071
0.8,49,15526,11970,414,14115
0.4,0,26415,20033,941,24475
0.4,1,19779,14184,964,15593
0.4,2,57368,51920,967,20384
0.4,3,19491,13912,969,14033
0.4,4,19954,14541,1002,16555
0.4,5,22300,17158,967,20124
0.4,6,57349,51112,956,19954
0.4,7,77611,73175,972,32961
0.4,8,23799,17497,942,20566
0.4,9,11020,9056,1014,11387
0.4,10,24705,18682,972,22504
0.4,11,22850,17272,913,21001
0.4,12,22211,17254,951,21015
0.4,13,72948,68515,934,31470
0.4,14,17852,13554,1021,15920
0.4,15,17362,14180,988,18209
0.4,16,19115,13766,1043,15427
0.4,17,14254,10017,971,10606
0.4,18,22111,16307,955,18415
0.4,19,71192,67865,949,23777
0.4,20,22844,16338,952,16808
0.4,21,13317,9768,962,10798
0.4,22,25355,19482,957,22380
0.4,23,26856,21363,915,25688
0.4,24,26407,20454,945,21278
0.4,25,11866,9347,980,12023
0.4,26,22143,15583,965,15884
0.4,27,33989,27201,932,32137
0.4,28,18288,13119,899,13782
0.4,29,37535,34690,964,13250
0.4,30,27690,21339,1001,26248
0.4,31,11931,8455,957,9281
0.4,32,31243,25016,976,29665
0.4,33,20104,14029,882,13354
0.4,34,27863,21569,983,26460
0.4,35,21829,16805,986,21537
0.4,36,117921,112411,957,45676
0.4,37,76567,71850,1005,26972
0.4,38,18689,14471,958,18670
0.4,39,30167,23299,942,28599
0.4,40,60036,54970,1009,23273
0.4,41,23007,16902,980,18076
0.4,42,23234,17179,890,19238
0.4,43,32573,28493,1040,10935
0.4,44,59320,53503,995,21066
0.4,45,27738,21237,932,23985
0.4,46,21376,16615,992,18416
0.4,47,17665,13011,969,13526
0.4,48,46359,42098,933,17204
0.4,49,20228,16120,859,19052
0.2,0,36636,27888,2614,32134
0.2,1,21548,15347,2631,17357
0.2,2,22514,16542,2521,19326
0.2,3,18525,13384,2491,16185
0.2,4,20652,14730,2512,17041
0.2,5,25358,18900,2583,22777
0.2,6,17264,12712,2506,15864
0.2,7,31082,24328,2535,29977
0.2,8,27275,20526,2549,24206
0.2,9,23135,18097,2595,23599
0.2,10,28576,22327,2553,27529
0.2,11,31904,25212,2774,30391
0.2,12,18040,12884,2579,15555
0.2,13,26531,20188,2643,24343
0.2,14,23873,17530,2492,20856
0.2,15,58841,52567,2654,35641
0.2,16,27789,20224,2588,21894
0.2,17,26953,20809,2561,25830
0.2,18,21216,15007,2541,16668
0.2,19,19830,14248,2583,16759
0.2,20,24393,18680,2537,23395
0.2,21,84101,78597,2579,29563
0.2,22,60118,55369,2555,22356
0.2,23,28910,23106,2541,27910
0.2,24,53028,47760,2641,24745
0.2,25,21713,16066,2700,19409
0.2,26,21541,16053,2610,18712
0.2,27,30215,22467,2598,26670
0.2,28,27665,20826,2545,24751
0.2,29,24716,18446,2616,23055
0.2,30,20882,15067,2549,17249
0.2,31,22285,16806,2544,22645
0.2,32,18555,13693,2575,16552
0.2,33,24749,18556,2554,22626
0.2,34,22926,17525,2497,22695
0.2,35,22617,16401,2757,19726
0.2,36,24081,18209,2535,22985
0.2,37,21531,16605,2601,22003
0.2,38,21839,16696,2502,21976
0.2,39,27377,20835,2532,25391
0.2,40,27098,20871,2544,26929
0.2,41,23579,17535,2701,22024
0.2,42,14942,10806,2630,13225
0.2,43,24212,18626,2631,22751
0.2,44,19636,13756,2618,15325
0.2,45,72963,67556,2567,48830
0.2,46,18237,12364,2608,13201
0.2,47,15604,11307,2500,13498
0.2,48,18184,13460,2521,17750
0.2,49,21668,16633,2727,22190
0.1,0,24154,18634,9339,31046
0.1,1,20052,14234,9468,23809
0.1,2,21756,16852,9609,29399
0.1,3,21707,17072,9639,30326
0.1,4,23658,18323,9642,31896
0.1,5,24938,19264,9640,32477
0.1,6,23177,17853,9625,29931
0.1,7,24963,20026,9495,32600
0.1,8,21407,16326,9413,28424
0.1,9,66136,61338,9823,34901
0.1,10,22678,17866,9578,30201
0.1,11,24918,19534,9671,32242
0.1,12,24101,18829,9673,32858
0.1,13,23473,18199,9419,31035
0.1,14,22901,17794,9527,29868
0.1,15,21745,16575,9353,28186
0.1,16,23721,18214,9658,30455
0.1,17,26461,21010,9454,33132
0.1,18,25113,19332,9853,32114
0.1,19,22283,17835,9407,31571
0.1,20,65262,60595,9447,34923
0.1,21,23479,18198,9549,30186
0.1,22,23561,18006,9495,29031
0.1,23,24579,18696,9607,30129
0.1,24,24928,19104,9243,29766
0.1,25,22026,16736,9647,28994
0.1,26,22519,17428,9342,30119
0.1,27,22668,17490,9495,29470
0.1,28,23691,18429,9438,29895
0.1,29,23201,18063,9402,30998
0.1,30,23649,18004,9475,30058
0.1,31,27056,20709,9636,31638
0.1,32,19097,13242,9689,21928
0.1,33,27162,21405,9672,34702
0.1,34,21911,17153,9542,28642
0.1,35,23240,18151,9622,30767
0.1,36,20274,15418,9585,27684
0.1,37,24555,19294,9365,31221
0.1,38,20467,15562,9593,28873
0.1,39,25273,19551,9600,31379
0.1,40,20915,16348,9527,28593
0.1,41,24100,19128,9523,32089
0.1,42,22336,17371,9717,31663
0.1,43,23687,18195,9670,30179
0.1,44,23619,17836,10085,30839
0.1,45,17490,12690,9613,22766
0.1,46,23319,18442,9815,32181
0.1,47,25184,19551,9399,31589
0.1,48,24020,18183,9671,29970
0.1,49,22263,17016,9500,29828
0.05,0,17611,14587,37320,57868
0.05,1,19771,16138,37227,58820
0.05,2,20334,16794,37438,59619
0.05,3,19191,15571,37929,58363
0.05,4,19128,15468,37404,57429
0.05,5,18900,15791,37082,59550
0.05,6,20630,16895,37647,59498
0.05,7,17417,14372,37238,55488
0.05,8,18727,15530,37644,58061
0.05,9,65230,62436,37632,65816
0.05,10,19014,15603,37227,58602
0.05,11,18225,15109,37447,57018
0.05,12,20160,16468,37290,58176
0.05,13,19161,15436,37507,57318
0.05,14,20824,17307,37201,59396
0.05,15,19240,15985,37228,58327
0.05,16,18983,15412,37389,56947
0.05,17,18257,14856,37337,56048
0.05,18,22510,18194,37644,60270
0.05,19,17765,13945,37487,55808
0.05,20,20120,16786,37731,59737
0.05,21,18124,14491,37638,56319
0.05,22,20717,17611,36827,59015
0.05,23,19184,15941,37037,58811
0.05,24,17806,14492,37768,57720
0.05,25,19047,15884,37468,59860
0.05,26,19072,15700,37282,58223
0.05,27,18333,14918,37221,56483
0.05,28,18669,15254,37385,56751
0.05,29,19506,15833,37249,56701
0.05,30,18074,14976,37278,56599
0.05,31,16781,13501,36958,55473
0.05,32,19192,15447,36703,56285
0.05,33,16728,13874,37079,55990
0.05,34,20125,17036,37029,60372
0.05,35,18495,14862,37535,57456
0.05,36,17676,14229,37524,56596
0.05,37,19699,15967,37450,57888
0.05,38,18328,14878,37262,58461
0.05,39,20314,16494,37522,59196
0.05,40,19335,15944,37216,57592
0.05,41,17676,14509,37158,56840
0.05,42,20443,17209,37349,62147
0.05,43,20149,15797,37423,57836
0.05,44,16109,13350,37332,55752
0.05,45,18544,15331,37828,57614
0.05,46,18143,14353,37246,56260
0.05,47,18917,14930,37515,56635
0.05,48,16203,13270,37095,56161
0.05,49,20952,17127,37737,60247

A box-plot

- Conveys
  - Four quartiles | Mean | Median | Outliers
  - Trend as a function of CI is clear as day



The R program

```
fh <- read.csv(file="~/Desktop/knap.csv",head=TRUE,sep=",")
ad <- vector()
ad <- append(ad,fh[1:50,][6]/1000)
ad <- append(ad,fh[51:100,][6]/1000)
ad <- append(ad,fh[101:150,][6]/1000)
ad <- append(ad,fh[151:200,][6]/1000)
ad <- append(ad,fh[201:250,][6]/1000)
# ad is now a vector of arrays

xl <- c("0.8","0.4","0.2","0.1","0.05")
mv <- numeric(0)
for(i in 1:5) {
  mv <- append(mv,mean(ad[i]$T));
}
# mv is now a vector of the means of each array
md <- numeric(0)
for(i in 1:5) {
  md <- append(md,median(ad[i]$T));
}
# md is now a vector of the medians of each array
sdv <- numeric(0)
for(i in 1:5) {
  sdv <- append(sdv,sd(ad[i]$T));
}
# sdv is now a vector of the standard deviations
```

The R program

```
width  <- 6   # width of chart in inches
height <- 4   # height of chart in inches

pdf(file="knap-ci-sensitivy.pdf",width=width,height=height,pointsize=12)
# the output device is a PDF file for inclusion in LaTeX
boxplot(ad,col="yellow",
        at=c(1,2,3,4,5),
        add=FALSE,
        cex.axis=0.8,
        cex.names=0.8,
        names=xl,
        log="y",
        xlab="CI",
        ylab="Time",
        title="Confidence Sensitivity")
                        # The statement above did the whole plot
lines(mv,col="red")     # add a red line for the mean
lines(md,col="blue")    # add a blue line for the median

legend("bottomright",c("mean","median"),col=c("red","blue"),bg="white",
       lty=1,cex=0.8)  # and finally, add a legend

dev.off()               # close the file, we are done
```

Advantages?

- Fully scriptable
  - The charts can be created from the script that runs the experiments!
  - Complete automation
  - No more issues redoing the results
  - Can tune the R script from the UI
  - Can also produce the LaTeX tables! (for use with \input)

Summary & Conclusion

- Experimental work is not that hard
- But
  - You must *carefully design* the experiment for a well formed question
  - You must be systematic
  - You must be disciplined
  - You must devote the resources (don't do it on a laptop you use!)
  - You ought to fully automate
  - You need a minimum of statistics

Above all

- Remember the objective
  - Experiments are there to convince your reader
  - Experiments are meant to be fully reproducible
- Take home message
  - Bad experiments are worse than no experiments
  - It is worth being systematic