Measuring Empirical Computational Complexity with `trend-prof`

Simon Goldsmith
Alex Aiken
Daniel Wilkerson

FSE 2007
September 7, 2007

Understanding Performance

- Existing tools
  - theoretical asymptotic complexity
    - e.g., big-$\mathcal{O}$ bounds, big-Θ bounds
  - empirical profiling
    - e.g., `gprof`
- We propose an “empirical asymptotic” tool
  - `trend-prof`

How does my code scale?

- Consider insertion sort
- Theoretical Asymptotic Complexity
  - worst case $\Theta(n^2)$
  - best case $\Theta(n)$
  - expected case depends on input distribution
- Empirical Profiling
  - e.g., 2% of total time
- trend-prof
  - empirically scales as, e.g., $n^{1.2}$

`trend-prof` measures workloads

- Run workloads and measure performance

	w_1
Workloads:	w_1
Block 1:	1
Block 2:	61
...	
Block 5:	1770
...	

`trend-prof`

- Run workloads and measure performance

Workloads:	w_1	w_2
Block 1:	1	1
Block 2:	61	201
...		
Block 5:	1770	19900
...		

trend-prof

- Run workloads and measure performance

Workloads:	w_1	w_2	...	w_{60}
Block 1:	1	1	...	1
Block 2:	61	201	...	60001
...				
Block 5:	1770	19900	...	1.79997e9
...				

trend-prof

- Look for performance trends in each block

Workloads:	w_1	w_2	...	w_{60}
Block 1:	1	1	...	1
Block 2:	61	201	...	60001
...				
Block 5:	1770	19900	...	1.79997e9
...				

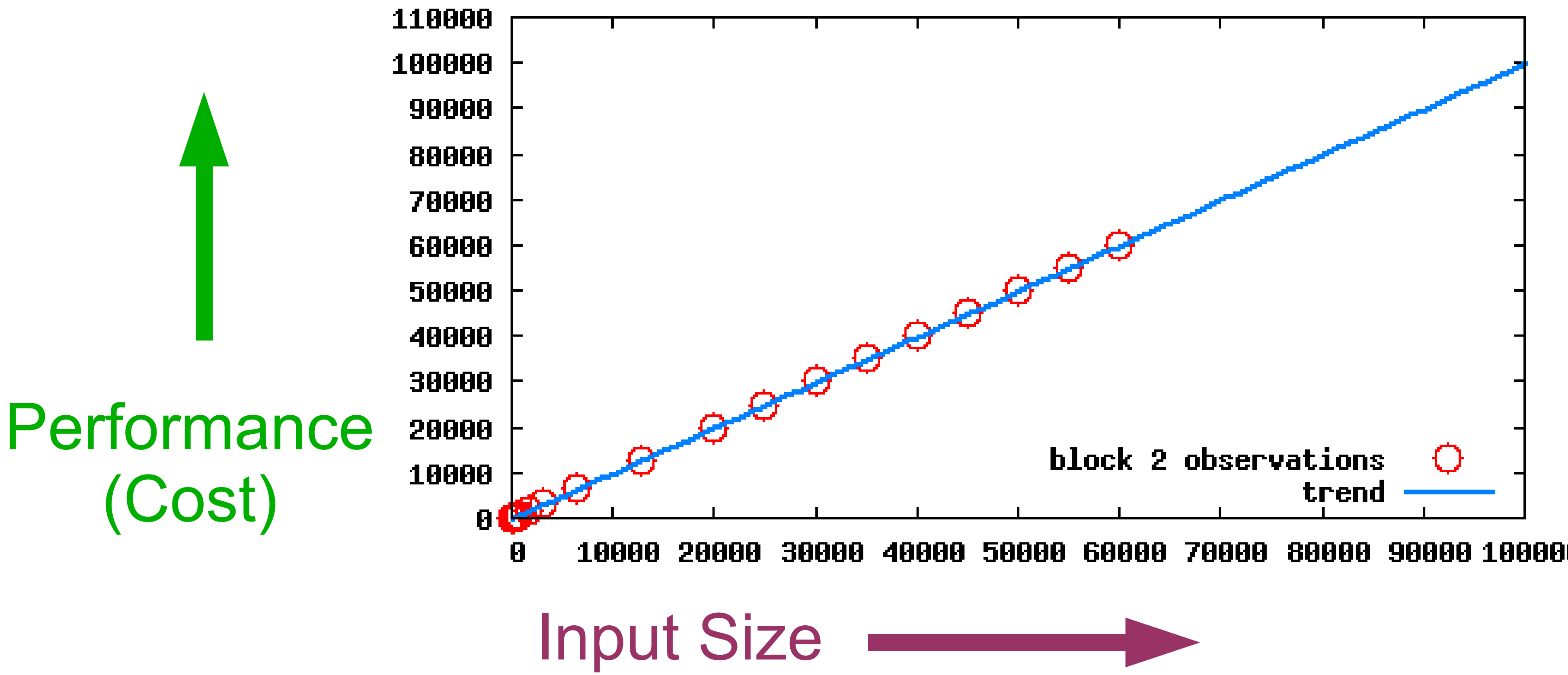
`trend-prof`: Input Size

- Look for performance trends in each block
  - with respect to user-specified input size

Workloads:	w_1	w_2	...	w_{60}
Input Size:	60	200	...	60000
Block 1:	1	1	...	1
Block 2:	61	201	...	60001
...				
Block 5:	1770	19900	...	1.79997e9
...				

Core Idea

- Relate performance of each basic block to input size



Uses of `trend-prof`

- Measure the performance trend an implementation exhibits on realistic workloads
  - and compare that to your expectations
- Identify locations that scale badly
  - may perform ok on smaller workloads, but dominate larger workloads

Example: `bsort`

```c
void bsort(int n, int *arr) {
  int i=0;
  while (i<n) {                   // O(n^2)
    int j=i+1;
    while (j<n) {                 // O(n^2)
      if (arr[i] > arr[j])
        swap(&arr[i], &arr[j]);
      j++; }
    ++i; } }
```

Challenges

- How to relate performance to input size?
- How to summarize a large amount of data?

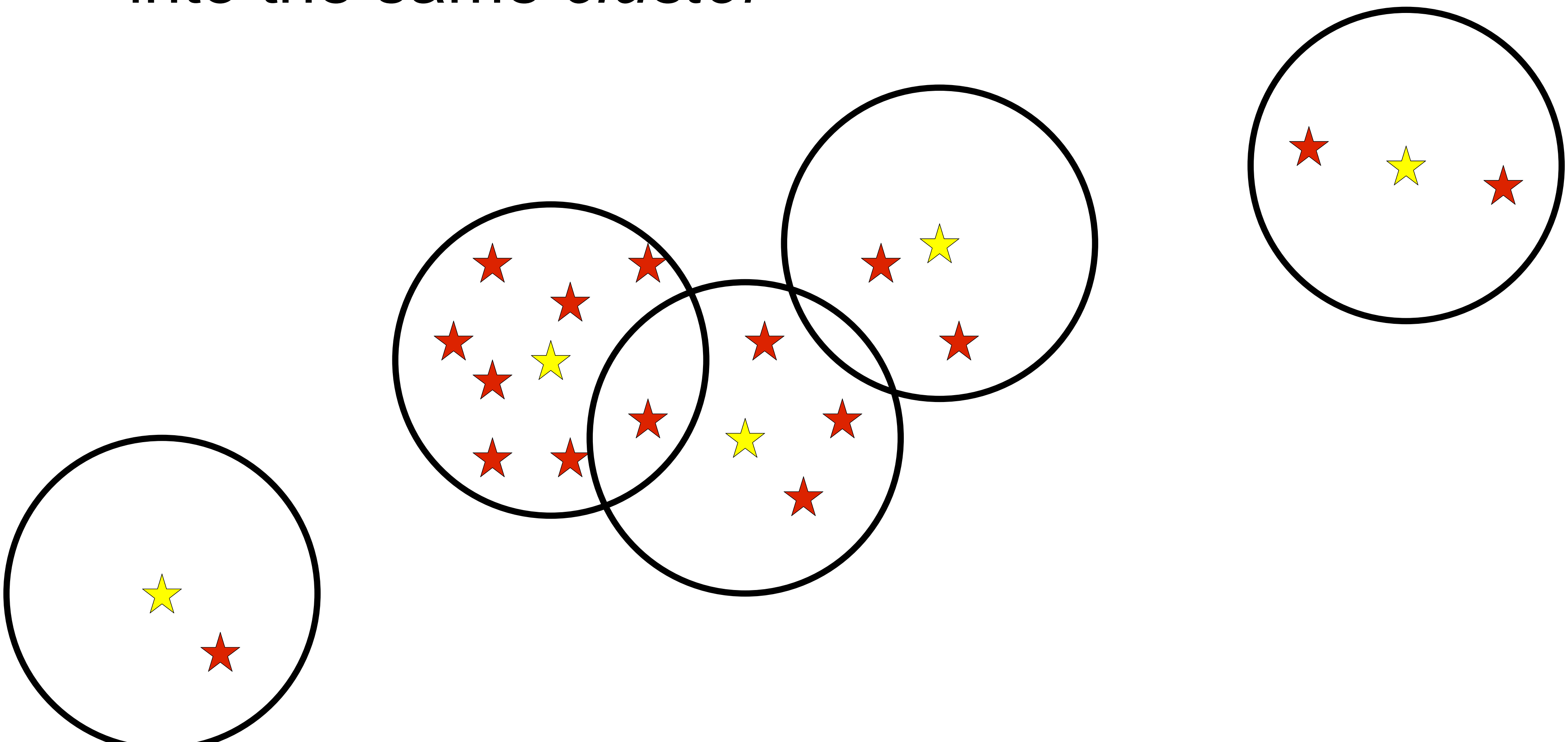
Problem: Too Many Basic Blocks

Program	Basic Blocks
`bzip`	1032
`maximus`	1220
`elsa`	33647
`banshee`	13308

- Leads to too many results to look at
  - Observation: Many basic blocks vary together

Summarize with Clusters

- Group basic blocks with similar performance into the same *cluster*



Empirical Fact: Clustering Works

Program	Basic Blocks	Clusters	Costly Clusters
`bzip`	1032	23	10
`maximus`	1220	13	9
`elsa`	33647	1489	30
`banshee`	13308	859	26

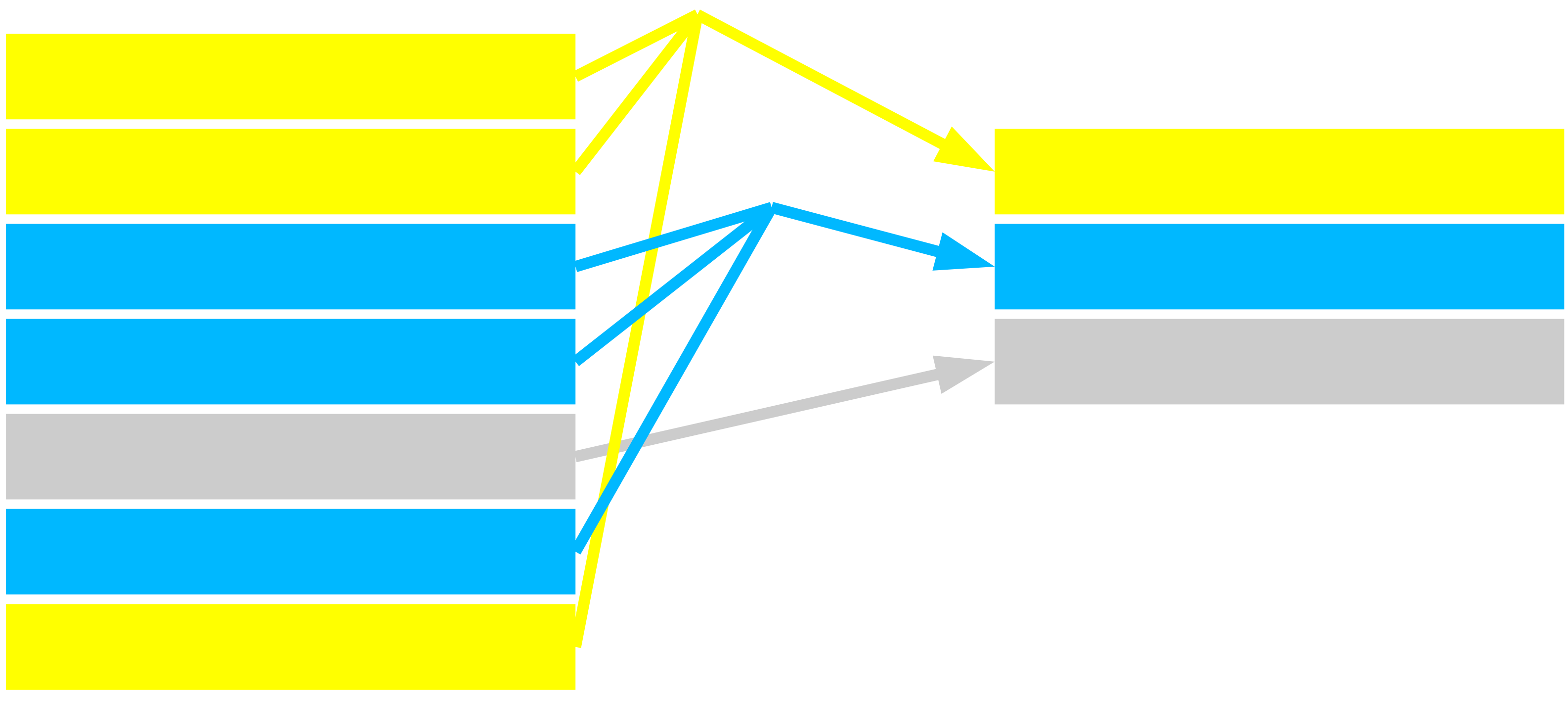
- Furthermore most clusters are small and cheap
  - a cluster is “costly” if it accounts for more than 2% of total performance on any workload

Clusters for bsort

```
void bsort(int n, int *arr) {
  int i=0;
  while (i<n) {
    int j=i+1;
    while (j<n) {
      if (arr[i] > arr[j])
        swap(&arr[i], &arr[j]);
      j++; }
    ++i; } }
```

Cluster Total as Matrix Row

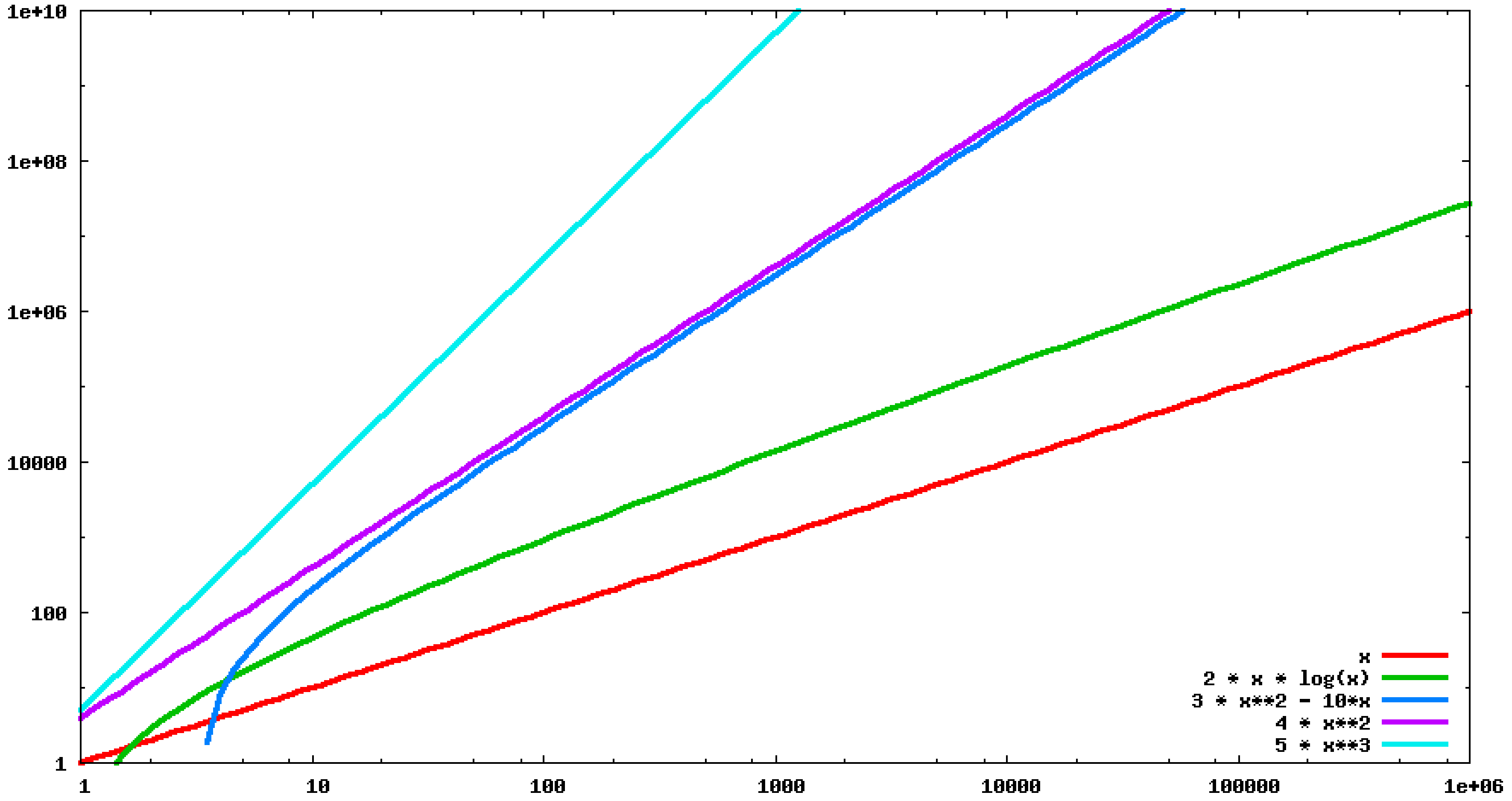
- Relate total executions of each cluster to input size



Relate Performance to Input Size

- Powerlaw regression is great
- (Cost) = a (Input Size)^b
  - Linear regression on (log Input Size, log Cost)
- Captures the high-order term
  - logarithmic factors don't matter in practice
  - polynomials converge to high order term

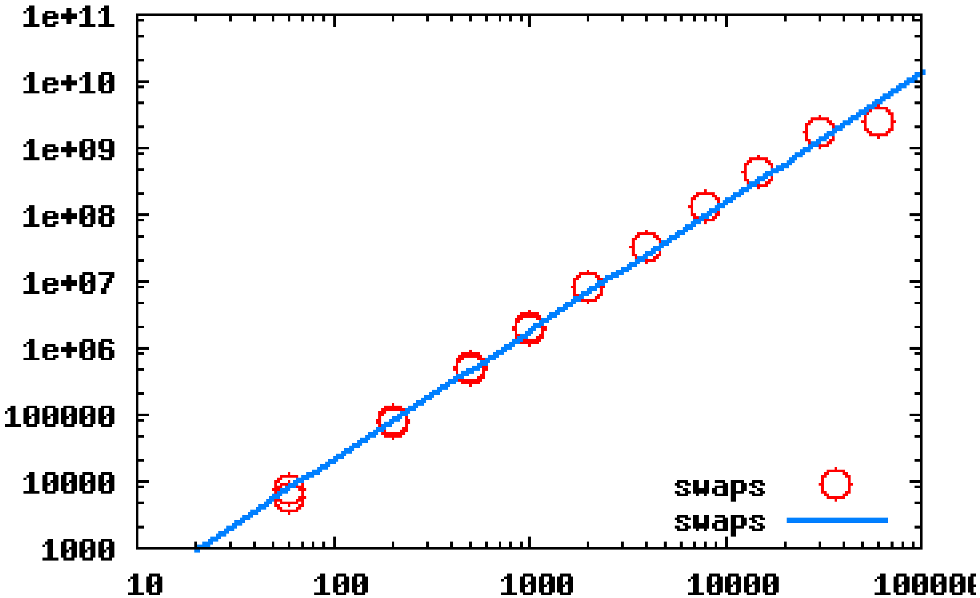
Powerlaw fit

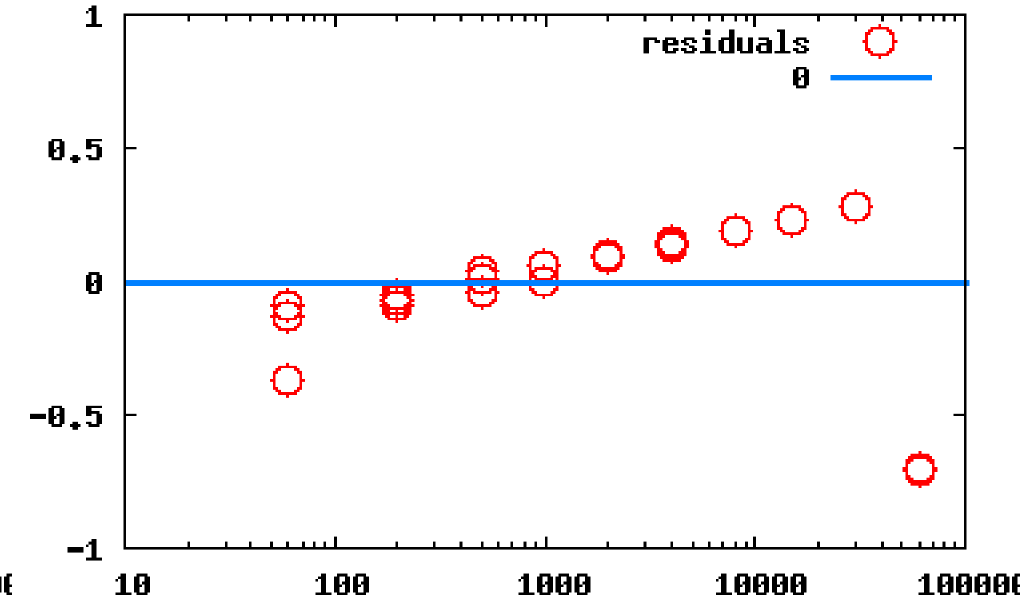


Output: bsort

max cost (billions of basic block executions)	Cluster	Cluster Total as a function of input size	R^2
11	Compares	$3.1\ n^{2.00}$	1.00
2.5	Swaps	$3.0\ n^{1.93}$	0.996
< 1	Size	$22\ n^{1.00}$	1.00

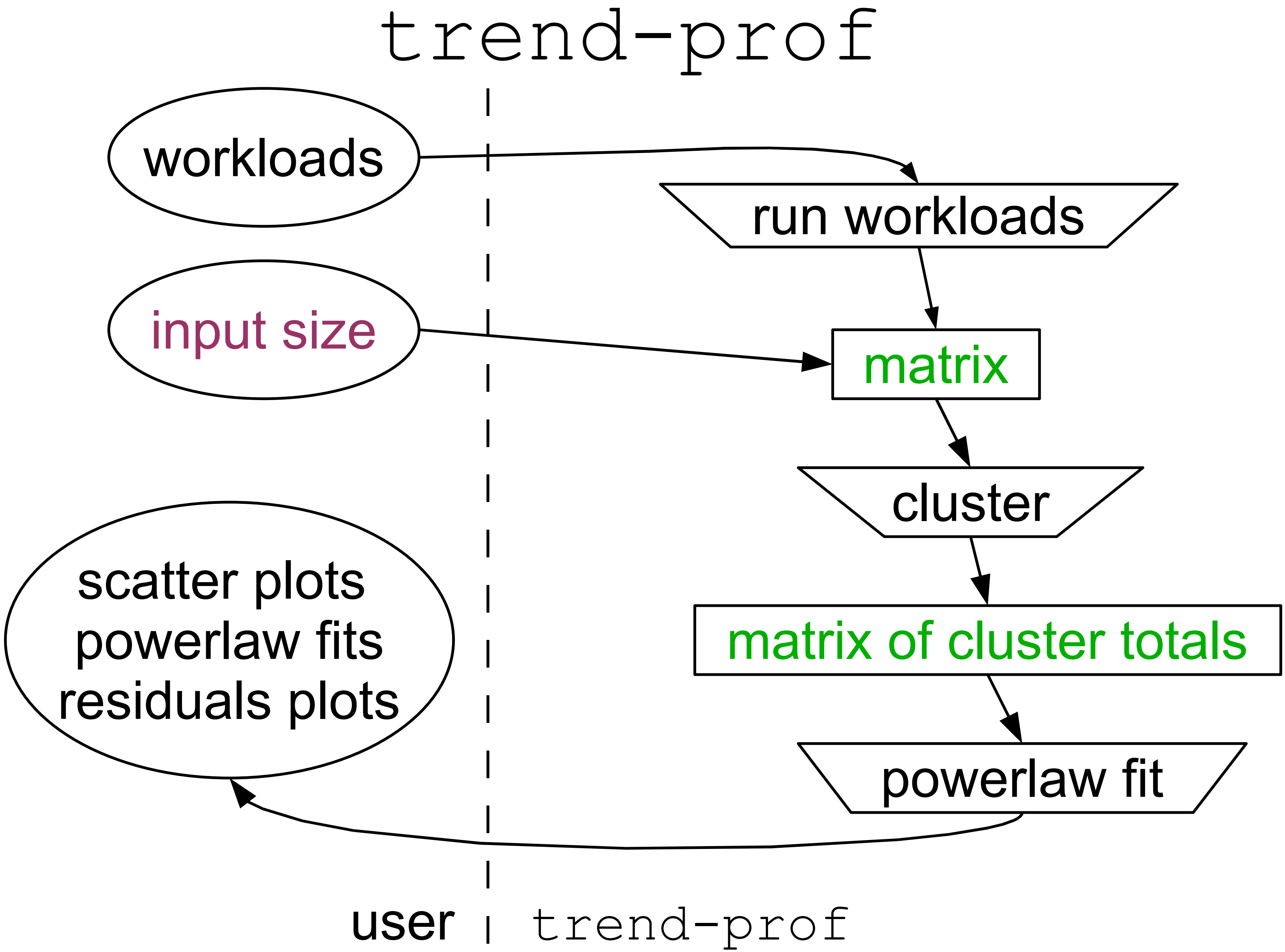
bsort: Plots





- log(size) vs log(swaps cluster)
- slope = 1.93

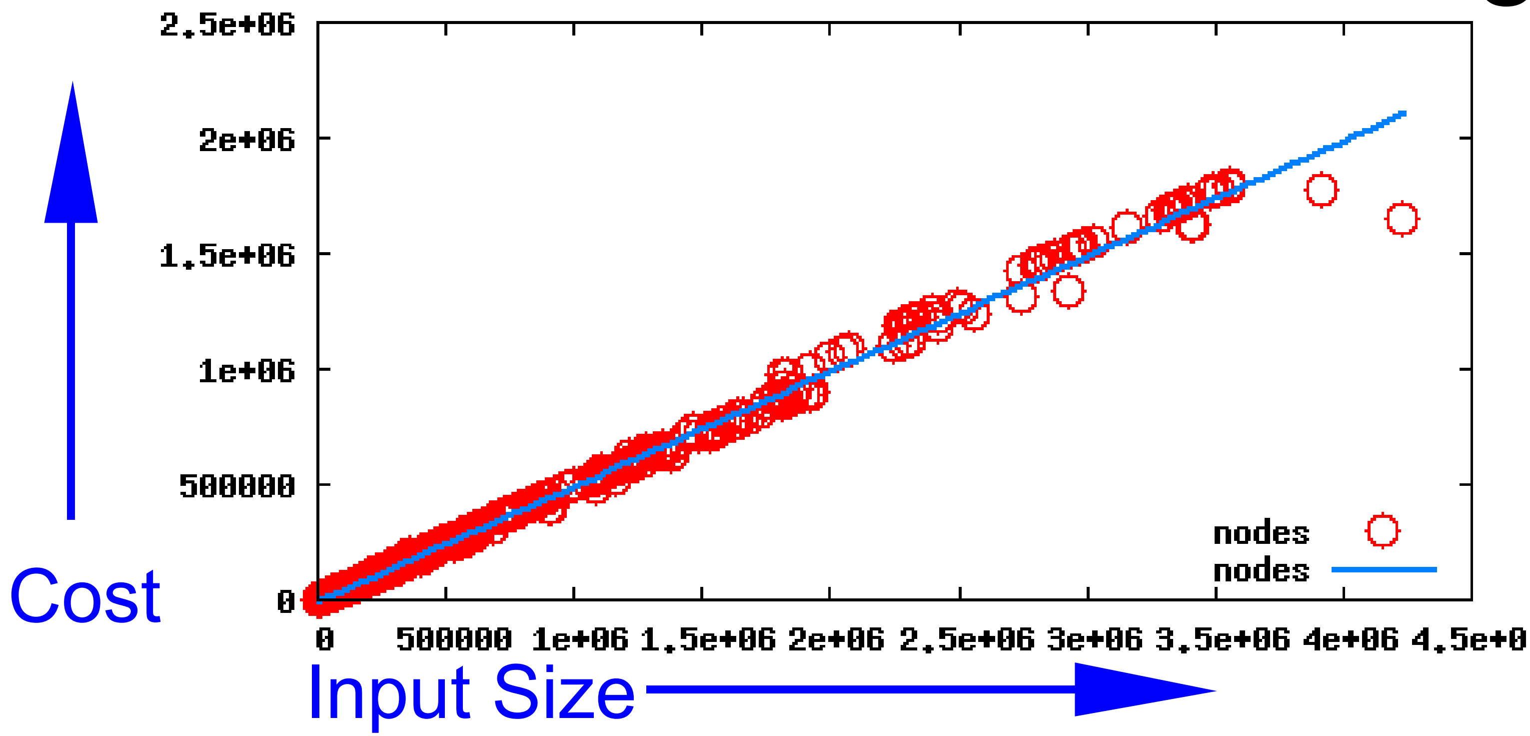
- residuals plot
  - they are small
  - they are not random



trend-prof
workloads
run workloads
input size
matrix
cluster
scatter plots
powerlaw fits
residuals plots
matrix of cluster totals
powerlaw fit
user
trend-prof

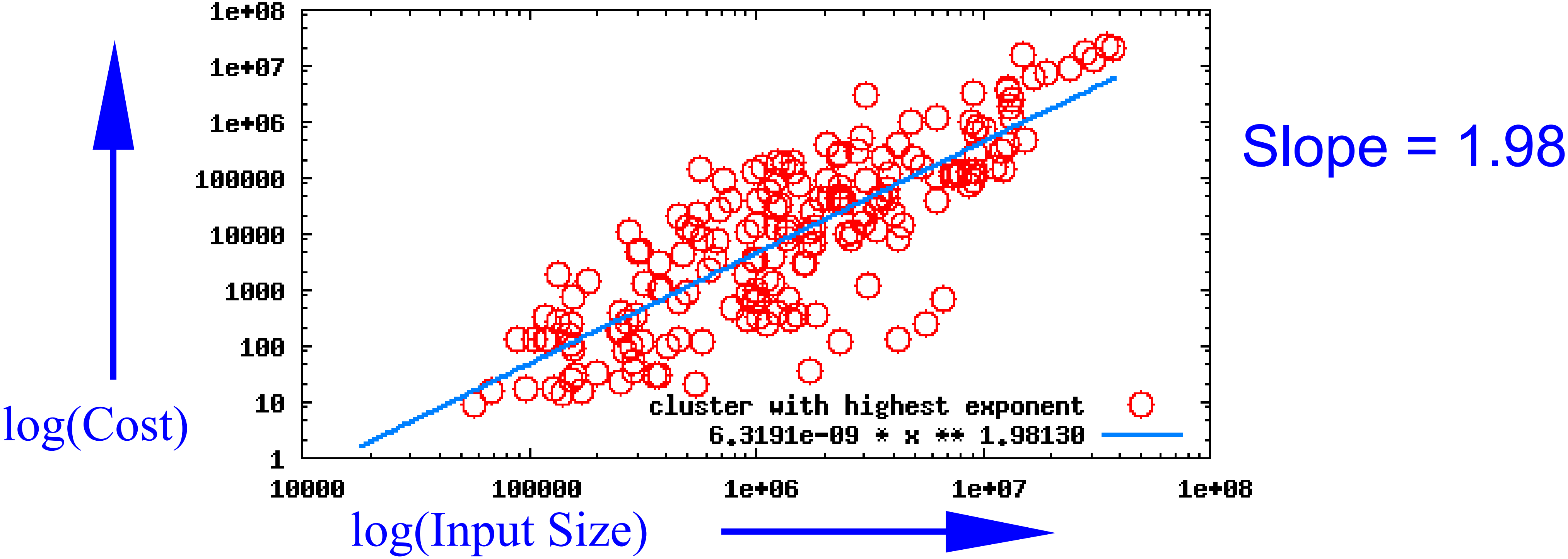
Results

Confirmed Linear Scaling



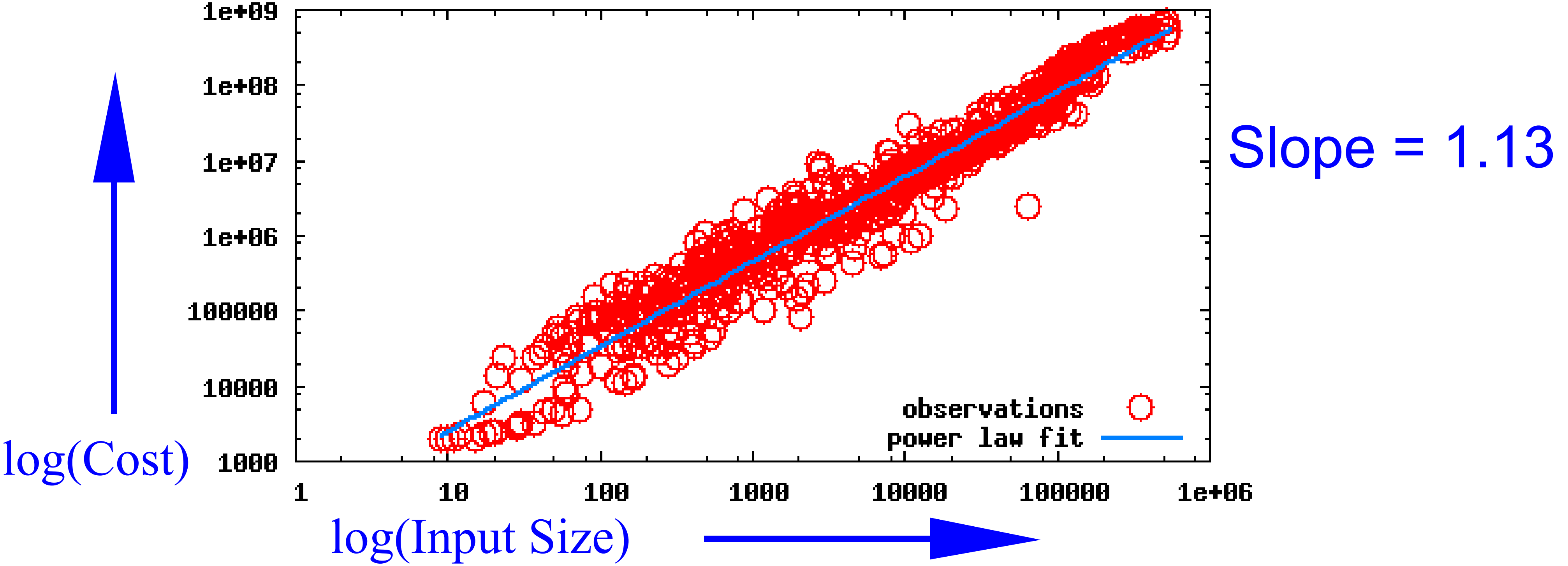
- Ukkonen's Algorithm (maximus)
  - Theoretical Complexity: O(n)
  - Empirical Complexity: ~ n

Empirical Complexity: Andersen's



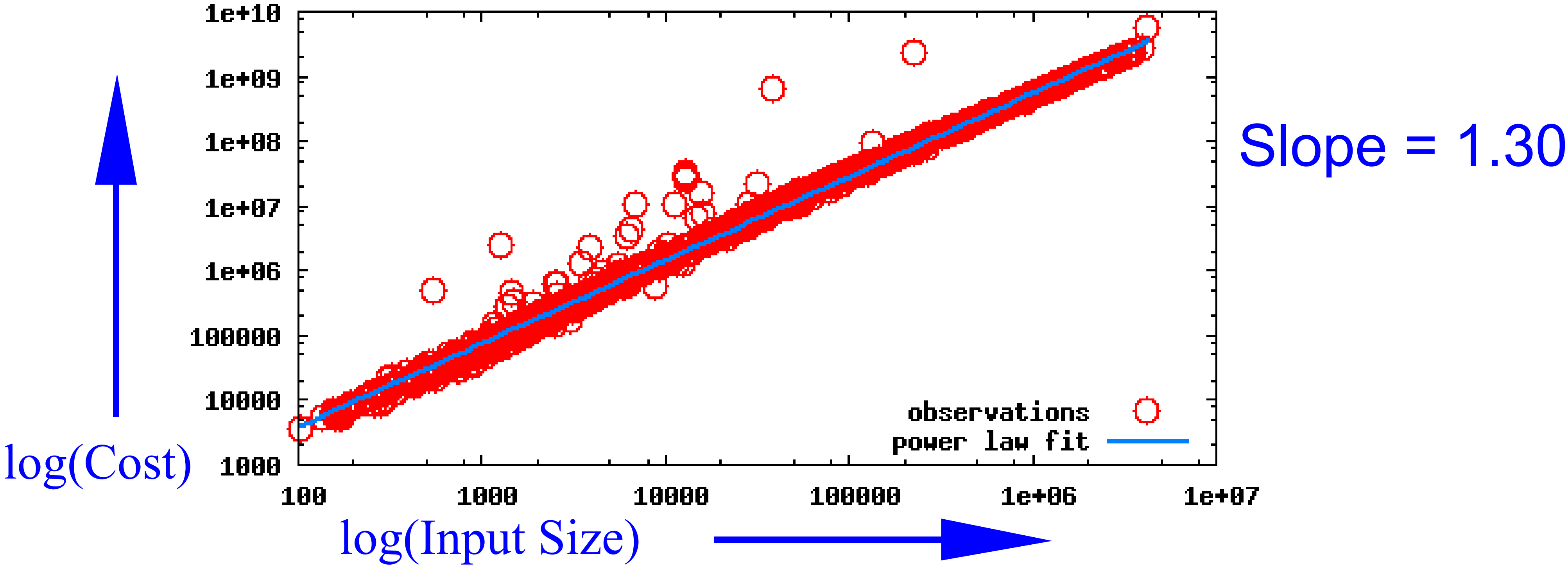
- Andersen's points-to analysis (banshee)
  - Theoretical Complexity: $O(n^3)$
  - Empirical Complexity: $\sim n^{1.98}$

Empirical Complexity: GLR



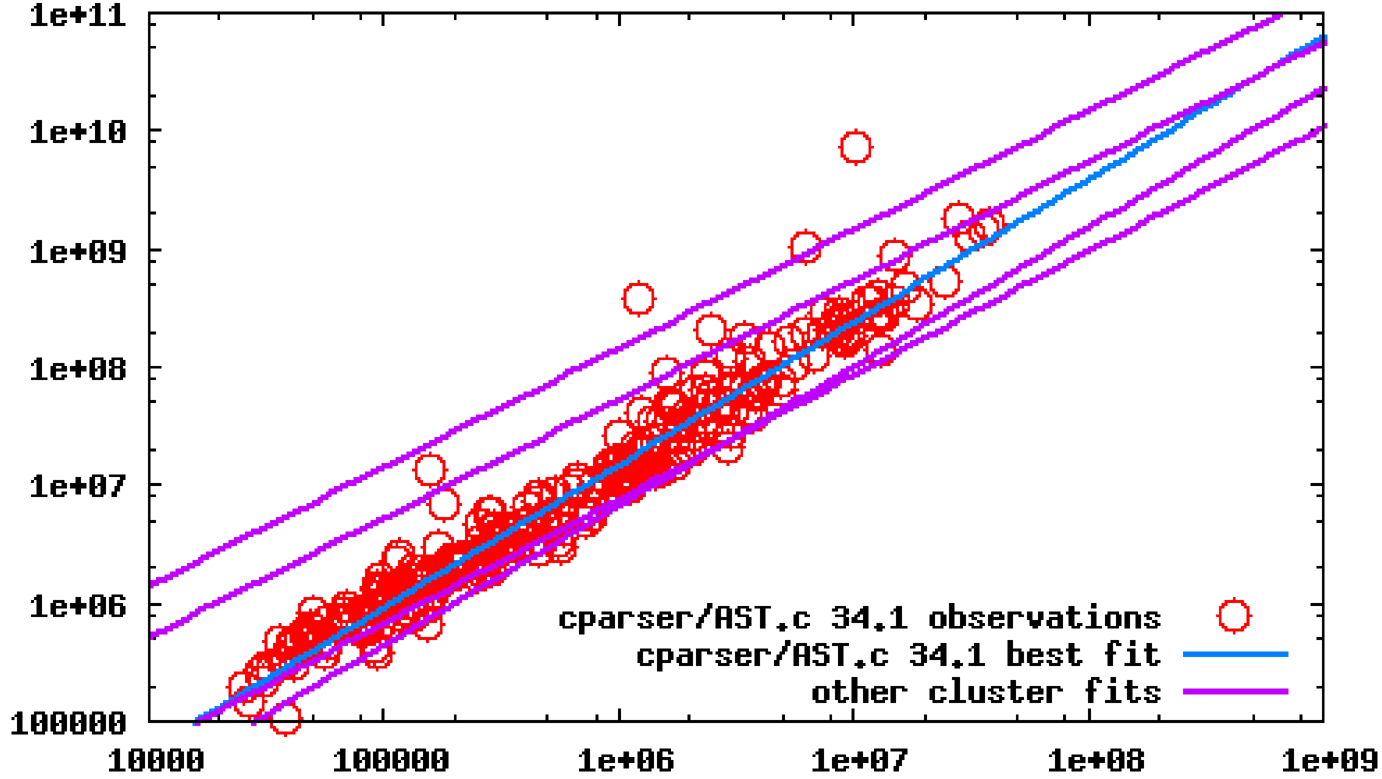
- GLR C++ parser (elkhound / elsa)
  - Theoretical Complexity: $O(n^3)$
  - Empirical Complexity: $\sim n^{1.13}$

How well do you know your code?



- Output routines (maximus)
  - Theoretical Complexity: O(n)?
  - Empirical Complexity: $\sim n^{1.30}$

Algorithms in context

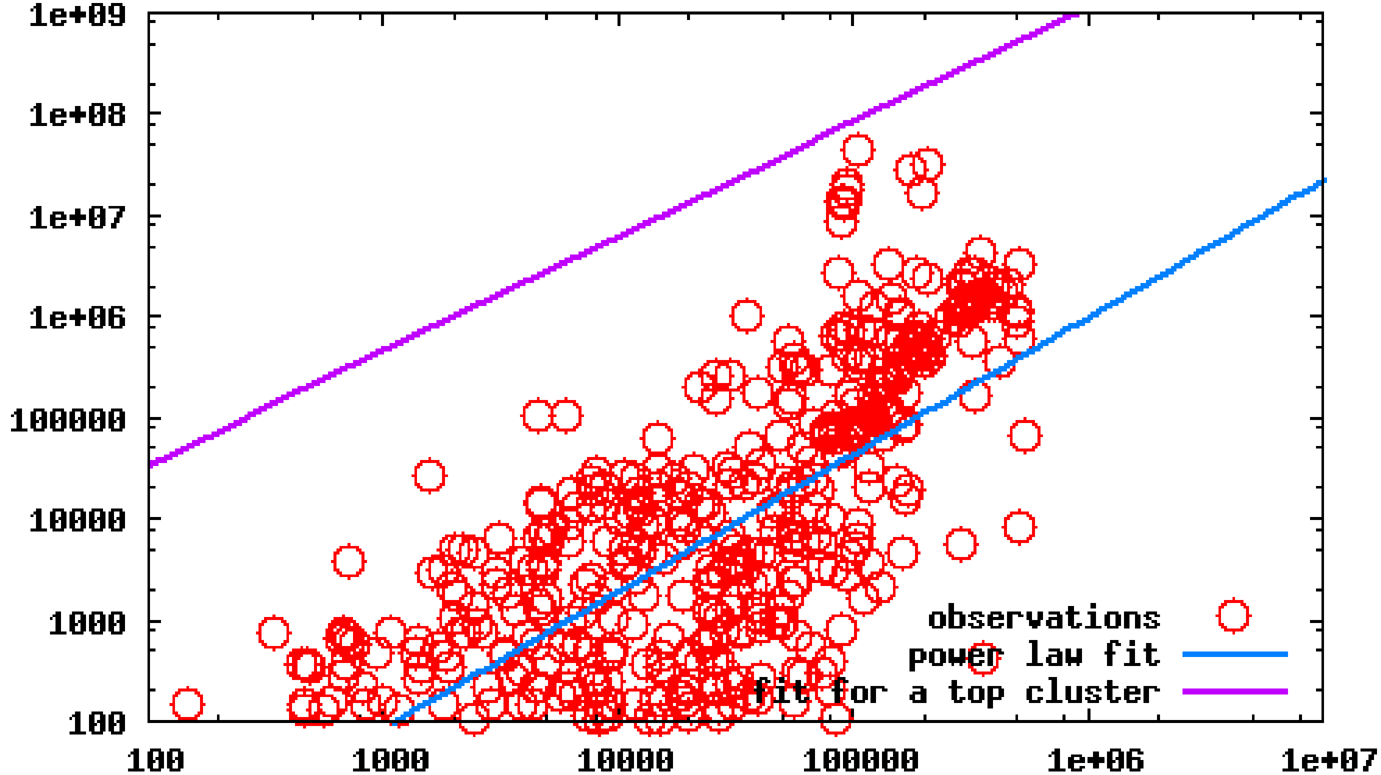


Slope = 1.21

$R^2 = 0.95$

- The linear-time list append in banshee's parser is a bug

Algorithms in Context



$R^2 = 0.65$

- The linear time list append in elsa's name lookup code is not a bug

Results Recap

- Confirmed linear scaling (maximus)
- Empirical scalability (Andersen's, GLR)
- Unexpected behavior (maximus)
- Algorithms in context (elsa, banshee)
  - found a performance bug in banshee's parser

Technical Contributions

- `trend-prof`
  - a tool to measure empirical computational complexity
- Discovery of the following empirical facts
  - programs have few costly clusters
  - powerlaw fits work well

Conclusion

- `trend-prof` models total basic block count of a cluster as a powerlaw function ($y = ax^b$) of user-specified input size
  - enables thorough comparison of your expectations about scalability to empirical reality
  - finds locations that scale badly

download `trend-prof` at http://trend-prof.tigris.org