Motivation

• Given an arbitrary data structure how can we determine the complexity of operations performed on it?
  – For example: consider a data structure that can store n elements.
    ➢ If a search is performed on the data structure what is the complexity of the operation ..(O(n)).
A sample experiment

• An Automatically Resizing array
  – Default array size n.
  – Array up sizes by n if maximum bound reached
  – Downsizes every time occupancy below 50%

• Can we determine this empirically
  – If a graph is plotted of operation performed vs. time taken, the graph should show a jump every time a resize operation is performed.
  – What if the data structure is cleverer, and the resize is performed based on the (I/D). Can we deduce this?

• Two Approaches
  – Empirical Analysis
    » Scales relatively better
    » Not very precise
  – Static Analysis
    » Gives a precise mathematical equation
    » Scales poorly with program size
Empirical Analysis

- Divide a program into basic blocks
- Define workloads to be analyzed
- Define features for each workload
  - Eg: For a bubble-sort algorithm, workload would be different sized arrays and features would be array sizes
- Measure performance of each basic block
- Combine blocks with highly correlated performance into clusters
- Curve fit the data points.
How is a Cluster Formed

• Consider a workload vector
  \[ W = \{ w_1, \ldots, w_n \} \]

• Described by features
  \[ f = \{ f_1, \ldots, f_k \}, g = \{ g_1, \ldots, g_k \} \]

• Compute execution counts for each program location \( l \)

• If counts for \( l \) fit existing cluster, add \( l \) to that cluster; if not, form a new cluster consisting of location \( l \).

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

Workloads: arrays of \( n \) integers (60, 200, 500, 1000...)
Feature: size of the array
<table>
<thead>
<tr>
<th>Cluster</th>
<th>Fit with n</th>
<th>R²</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPARSES &lt;4,5,7&gt;</td>
<td>3.0 n^2.00</td>
<td>1.00</td>
</tr>
<tr>
<td>SWAPS &lt;6&gt;</td>
<td>3.1 n^1.93</td>
<td>0.99</td>
</tr>
<tr>
<td>SIZE &lt;2,3,8&gt;</td>
<td>22 n^1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

**Limitations**

- We need to be careful about the types of workloads selected.
  - For example in the bubble sort algorithm, if the input workloads were already sorted, some paths in the code would never be executed.
- The power law curve cannot describe all complexities. For instance if the complexity is nlogn, it cannot be described by a logn curve fit.
- The curve fit converges towards the higher order terms. For example if a cluster varies as 1.5n^2 + 0.5n, the curve produced is for 1.5n^2. This works well for large workloads but is inaccurate for smaller workloads.
Static Analysis

- The hard part of computing bounds for programs is to compute bounds on loops.
  - Once we have bounds on loops we can compose these together to obtain a bound for the entire procedure.
- I will be talking about a couple of ways to compute bounds on loops
  - Instrumenting loops with counter variables.
    - Single Counter Instrumentation
    - Multiple Counter Instrumentation
Single Counter Instrumentation

Input: m
x := 0; y := 0;
i := 0;
while (x < 100)
i := i + 1;
if (y < m)
y := y + 1;
else
x := x + 1;

• First we need to compute an invariant that relates i to program variables.
  — This can be done using program abstraction techniques.
• The invariant that we would generate would be
  — i <= x + y + 1 \land y <= \text{Max}(0,m) \land x < 100
• Eliminate temporary variables
  — i <= 100 + \text{Max}(0,m)
• This gives us the required bound on the loop, 100 + \text{Max}(0,m)
Multiple Counter Instrumentation

Inputs: n,m
x := 0; y := 0;
\(i_1 := 0; i_2 := 0;\)
while (x < 100)
  if (y < m)
    y := y + 1;
    \(i_1 := i_1 + 1;\)
  else
    x := x + 1;
    \(i_2 := i_2 + 1;\)

- Instrument loop with multiple counters
- Compute invariants for all counters
  - \(i_1 \leq \text{Max}(0,m)\)
  - \(i_2 \leq \text{Max}(0,100)\)
- Thus bound on entire loop is
  - \(= i_1 + i_2 = 100 + \text{Max}(0,m)\)
- Easier to generate invariants (it is possible to use linear invariant generation tools for this purpose)
- Where we initialize and increment counters is trickier
Program Transformation

Inputs: \( m \)
x := 0; y := 0;
while (x < 100)
  if (y < m)
    y := y + 1;
  else
    x := x + 1;

Questions?