SAPIENZA
UNIVERSITÀ DI ROMA

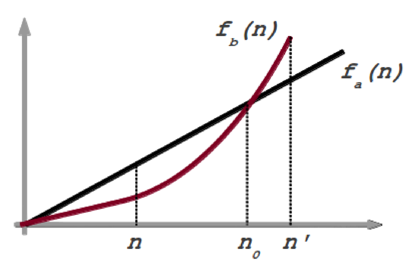
Input-Sensitive Profiling

(or how to find the big-Oh of a program?)

Emilio Coppa, Camil Demetrescu, and Irene Finocchi

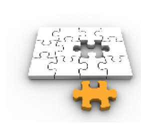
http://code.google.com/p/aprof/

Conventional profilers collect cumulative data over a whole execution...



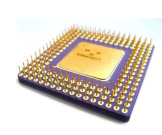
No information about how performance of **single portions** of code **scales** as a function of the **input size**

A possible approach is to extract and isolate the interesting code and perform multiple under a traditional profiler with different input but...



often hard to isolate portions of code and analyze them separately...

Hard to collect real data about typical usage scenarios...



Miss cache effects due to the interaction with the overall application...

Input-Sensitive Profiling: aggregate routine times by input sizes

For routine f, collect a set of tuples, where each tuple contains:

- an estimate of an input size
- number of invocations on this input size
- max/min/avg execution cost

We need a metric for estimating the input size of a routine invocation...

How can measure the input size of a routine invocation **automatically**?

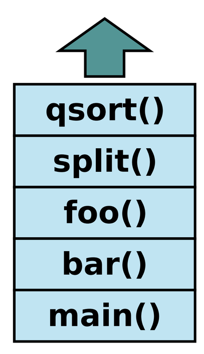
Read Memory Size: number of distinct memory cells first accessed by a routine, or by a descendent in the call tree, with a read operation

```
call f
  read x
  write y
  call g
    read x
    read y
    read z
    write w
    return
  read w
  return
```

RMS(f) = 2
RMS(g) = 3

How can we compute **efficiently** the read memory size?

Two data structures:



1) a **shadow runtime stack**, where each entry contains:

- ID of pending routine
- routine entry timestamp
- total routine invocation cost
- **partial read memory size**

- more efficient/compact
- equal to the RMS upon invocation completion

2) a **shadow memory**:

t_x	t_y	t_z
x	y	z

For each memory location w, timestamp ts[w] contains the time of **latest** access (read or write) to w

Profiling algorithm:

```
procedure call(r):
  top++
  S[top].rtn ← r
  S[top].ts ← ++count
  S[top].rms ← 0
  S[top].cost ← get_cost()
```

```
procedure return():
  collect(S[top].rtn, S[top].rms,
          get_cost() -S[top].cost)
  S[top − 1].rms += S[top].rms
  top−−
```

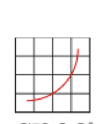
```
procedure read(w):
  if ts[w ] < S[top].ts then
    S[top].rms++
    if ts[w] = 0 then
      let i be the max index in S
      such that S[i].ts ≤ ts[w]
      S[i].rms−−
    end if
  end if
  ts[w] ← count
```

```
procedure write(w):
  ts[w] ← count
```

aprof

input-sensitive profiler based on

Comparable performance wrt other Valgrind tools. Experiments on CPU SPEC 2006 suite:



slowdown: ~30x
space overhead: ~2x

Profile data generated by aprof from a **single run** would require multiple runs of gprof

Case study: wf

We discuss **wf,** a simple word frequency counter included in the current development head of Fedora Linux.

Our goal: study how the perfomance of individual routines scales as a function of the input size. To do so, for each routine of wf, we plot a chart with k points.

We analyze wf with:

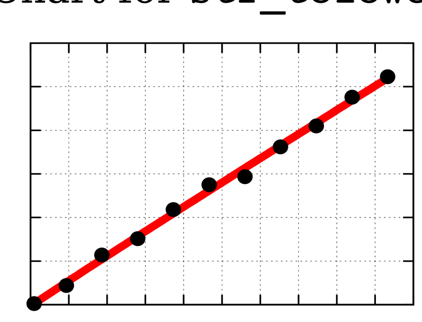
gprof

For each point of a chart we need to perform a separate run of wf.

1 run = 1 point

Input of wf: texts of increasing size from classical literature

Chart for str_tolower



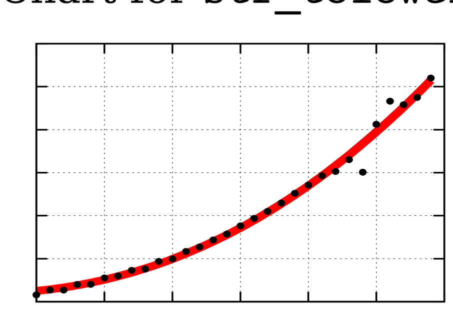
aprof

aprof can collects several points for a chart from the same execution of a program by aggregating routine times by input sizes

1 run = N points

Input of wf: smallest text used with gprof

Chart for str_tolower



Linear growth vs quadratic growth
which one is correct?

strlen() redundantly called at each iteration: $O(n^2)$

```
void str_tolower(char* str) {
  int i;
  for (i = 0; i < strlen(str); i++)
    str[i] = wf_tolower(str[i]);
}
```

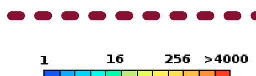
Fix the code by loop-invariant code motion:

```
void str_tolower(char* str) {
  int i, len = strlen(str);
  for (i = 0; i < len; i++)
    str[i] = wf_tolower(str[i]);
}
```

Performance improvement of wf up to 30%

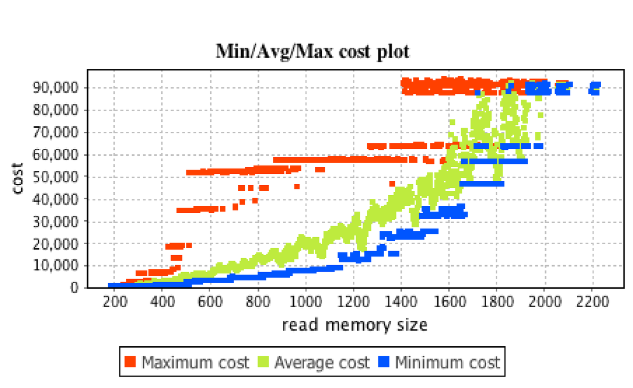
Lesson: input of str_tolower are single words, not the entire text. aprof automatically measures cost for each distinct word length.

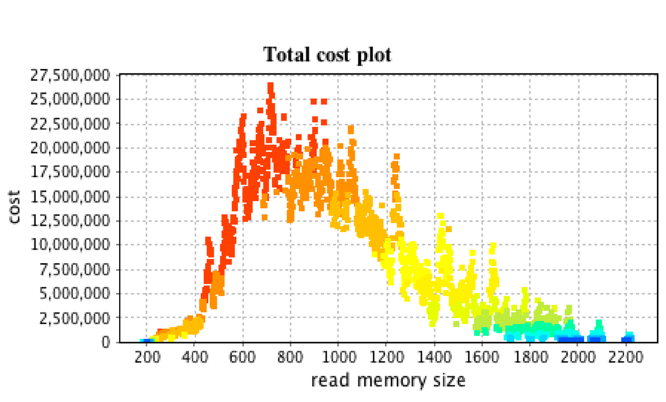
Profiles of CPU SPEC 2006 benchmarks: examples



tonto: quantum chemistry

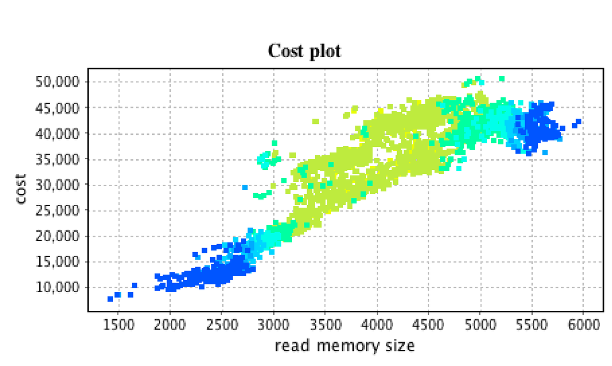
__shell1quartet_module__make_r_jk_ascd()





gobmk: artificial intelligence

owl_shapes()



h264ref: video compression

PartitionMotionSearch()