SAPIENZA
UNIVERSITÀ DI ROMA

Input-Sensitive Profiling

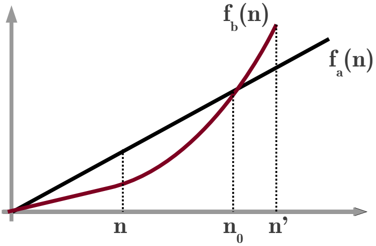
Emilio Coppa
Camil Demetrescu
Irene Finocchi

June 11, 2012

PLDI 2012

Conventional profilers

Conventional profilers gather **cumulative info** over a whole execution



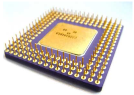
$\Longrightarrow$ No information about how single portions of the code **scale** as a function of **input size**

Drawbacks classical approach

- Often hard to extract portions of code from an application and analyze them separately

- Hard to collect real data about typical usage scenarios to be reproduced in experiments

- Miss cache effects due to interaction with the overall application



Critical algorithmic code should be analyzed within the **actual** context of applications it is deployed in

Our approach

"Input-Sensitive" profiling: aggregating routine times by input sizes

For each routine f, collect a tuple:

$$< n_i, c_i, max_i, min_i, sum_i, q_i >$$

for each distinct value of the input size, where:

- n_i = estimate of an input size
- $c_i = \#$ of times the routine is called on input size n_i
- max_i/min_i = maximum and minimum costs required by any execution of f on input size n_i
- sum_i/q_i = sum of the costs required by the executions of f on input size n_i and the sum of the costs' squares

How to measure input size automatically?

Input size ≈ Read Memory Size

The **read memory size** (RMS) of the execution of a routine f is the number of distinct memory cells first accessed by f, or by a descendant of f in the call tree, with a read operation.

Read Memory Size (Example)

```
void swap(int * a, int * b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

The function `swap` has RMS 2 because it reads (first access) objects `*a` and `*b`, and writes (first access) variable `temp`

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)	RMS

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)	RMS
f		

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)	RMS
f	x	1

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)	RMS
f	x y	1

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)	RMS
f	x y	1
g		

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)	RMS
f	x y	1
g	x	1

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)	RMS
f	x y	1
g	x y	2

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)	RMS
f	x y z	2
g	x y z	3

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)				RMS
f	x	y	z	w	2
g	x	y	z	w	3

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)				RMS
f	x	y	z	w	2
g	x	y	z	w	3

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)				RMS
f	x	y	z	w	2
g	x	y	z	w	3

Read Memory Size (Example 2)

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

Fn	Accessed cells (first-read green)				RMS
f	x	y	z	w	2
g	x	y	z	w	3

Case study: discovering asymptotic inefficiencies

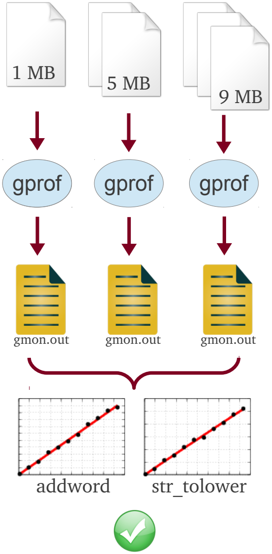
We discuss `wf-0.41`, a simple word frequency counter included in the current development head of Linux Fedora (Fedora 17–Beefy Miracle).

We profile `wf` with:

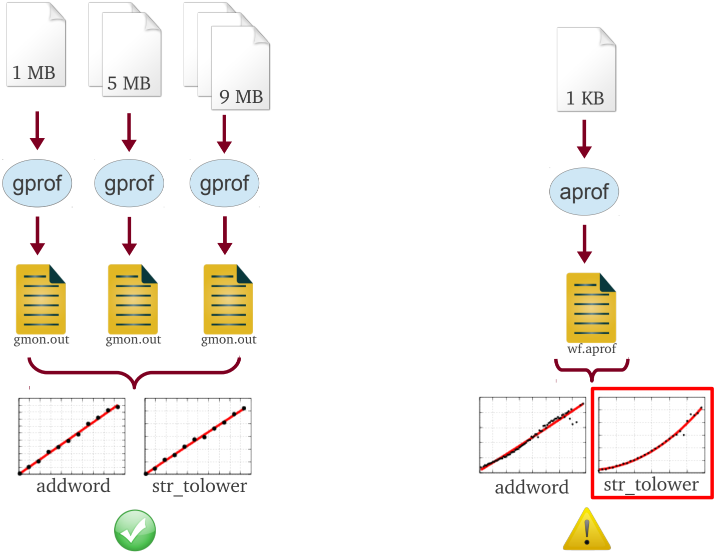
gprof
: a traditional and well-known call graph execution profiler – http://www.gnu.org/software/binutils/

aprof
: asymptotic profiler
our implementation of an input-sensitive profiler – http://code.google.com/p/aprof/

We discuss `wf-0.41`: gprof vs aprof



We discuss `wf-0.41`: gprof vs aprof



Is there any bottleneck in str_tolower?

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

Is there any bottleneck in str_tolower?

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

Is there any bottleneck in str_tolower?

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

Why did gprof **fail** to reveal the quadratic trend of `str_tolower`?

Short vs long words in gprof

Input of `str_tolower` = single words of input text **not** the input text!

Short vs long words in gprof

Input of `str_tolower` = single words of input text **not** the input text!

Input: Anna Karenina

52.2%	`addword`
31.3%	`str_tolower`

Input: Protein sequences

61.8%	`str_tolower`
32.6%	`addword`

Short vs long words in gprof

Input of `str_tolower` = single words of input text **not** the input text!

Input: Anna Karenina

52.2%	`addword`
31.3%	`str_tolower`

Input: Protein sequences

61.8%	`str_tolower`
32.6%	`addword`

- Need to have different workloads for different routines!
- How do we know in advance which routine is a bottleneck?

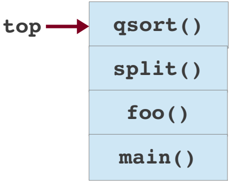
Fixing the code

Loop invariant code motion:

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

Improvements:	6%	Anna Karenina
	30%	Protein sequences

Computing RMS: data structures



Shadow run-time stack S

For each $i \in [0, top]$, the i-th stack entry $S[i]$ stores:

- *rtn*: id of the routine
- *ts*: timestamp assigned to this activation
- *cost*: cumulative cost
- *rms*: *partial read memory size* of the activation

Each memory location w has a timestamp $ts[w]$ which contains the time of the **latest access** to w

Computing RMS (example)

Step 0: `count = 0`

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

Fn	RMS	Cost
g	?	?
f	?	?

SHADOW RUNTIME STACK

TIMESTAMPS

```
ts[x] = 0
ts[y] = 0
ts[z] = 0
ts[w] = 0
```

Computing RMS (example)

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

Fn	RMS	Cost
g	?	?
f	?	?

Step 1: `count = 1`

SHADOW RUNTIME STACK

```
S[0].id = f
S[0].cost = 1
S[0].ts = 1
S[0].rms = 0
```

TIMESTAMPS

```
ts[x] = 0
ts[y] = 0
ts[z] = 0
ts[w] = 0
```

Computing RMS (example)

Step 2: `count = 1`

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

Fn	RMS	Cost
g	?	?
f	?	?

SHADOW RUNTIME STACK

```
S[0].id = f
S[0].cost = 1
S[0].ts = 1
S[0].rms = 1
```

TIMESTAMPS

```
ts[x] = 1
ts[y] = 0
ts[z] = 0
ts[w] = 0
```

Computing RMS (example)

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

Fn	RMS	Cost
g	?	?
f	?	?

Step 3: `count = 1`

SHADOW RUNTIME STACK

```
S[0].id = f
S[0].cost = 1
S[0].ts = 1
S[0].rms = 1
```

TIMESTAMPS

```
ts[x] = 1
ts[y] = 1
ts[z] = 0
ts[w] = 0
```

Computing RMS (example)

Step 4: `count = 2`

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

Fn	RMS	Cost
g	?	?
f	?	?

SHADOW RUNTIME STACK

```
S[1].id = g
S[1].cost = 4
S[1].ts = 2
S[1].rms = 0

S[0].id = f
S[0].cost = 1
S[0].ts = 1
S[0].rms = 0
```

TIMESTAMPS

```
ts[x] = 1
ts[y] = 1
ts[z] = 0
ts[w] = 0
```

Computing RMS (example)

Step 5: `count = 2`

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

Fn	RMS	Cost
g	?	?
f	?	?

SHADOW RUNTIME STACK

```
S[1].id = g
S[1].cost = 4
S[1].ts = 2
S[1].rms = 3

S[0].id = f
S[0].cost = 1
S[0].ts = 1
S[0].rms = 1-2 = -1
```

TIMESTAMPS

```
ts[x] = 2
ts[y] = 2
ts[z] = 2
ts[w] = 0
```

Computing RMS (example)

Step 6: count = 2

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

Fn	RMS	Cost
g	?	?
f	?	?

SHADOW RUNTIME STACK

```
S[1].id = g
S[1].cost = 4
S[1].ts = 2
S[1].rms = 3
```

```
S[0].id = f
S[0].cost = 1
S[0].ts = 1
S[0].rms = -1
```

TIMESTAMPS

```
ts[x] = 2
ts[y] = 2
ts[z] = 2
ts[w] = 2
```

Computing RMS (example)

Step 7: `count = 2`

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

Fn	RMS	Cost
g	3	5
f	?	?

SHADOW RUNTIME STACK

```
S[0].id = f
S[0].cost = 1
S[0].ts = 1
S[0].rms = -1+3 = 2
```

TIMESTAMPS

```
ts[x] = 2
ts[y] = 2
ts[z] = 2
ts[w] = 2
```

Computing RMS (example)

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

Fn	RMS	Cost
g	3	5
f	?	?

Step 8: `count = 2`

SHADOW RUNTIME STACK

```
S[0].id = f
S[0].cost = 1
S[0].ts = 1
S[0].rms = 1
```

TIMESTAMPS

```
ts[x] = 2
ts[y] = 2
ts[z] = 2
ts[w] = 2
```

Computing RMS (example)

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

Fn	RMS	Cost
g	3	5
f	2	10

Step 9: `count = 2`

SHADOW RUNTIME STACK

TIMESTAMPS

```
ts[x] = 2
ts[y] = 2
ts[z] = 2
ts[w] = 2
```

Computing RMS: algorithm

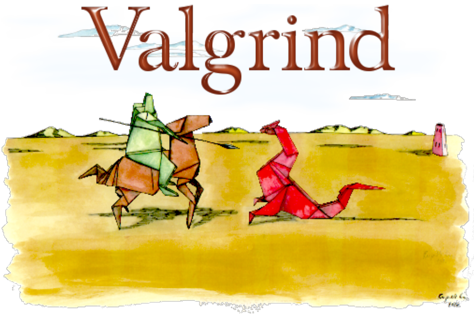
procedure `call`(r): $O(1)$
  $top{+}{+}$
  $S[top].rtn \leftarrow r$
  $S[top].ts \leftarrow {+}{+}count$
  $S[top].rms \leftarrow 0$
  $S[top].cost \leftarrow$ `get_cost()`

procedure `return`(): $O(1)$
  `collect`($S[top].rtn$, $S[top].rms$,
    `get_cost()` $- S[top].cost$)
  $S[top-1].rms \mathrel{+}= S[top].rms$
  $top{-}{-}$

procedure `read`(w): $O(\log(stack\ depth))$
  if $ts[w] < S[top].ts$ **then**
    $S[top].rms{+}{+}$
    if $ts[w] \neq 0$ **then**
      let i be the max index in S
      such that $S[i].ts \leq ts[w]$
      $S[i].rms{-}{-}$
    end if
  end if
  $ts[w] \leftarrow count$

procedure `write`(w): $O(1)$
  $ts[w] \leftarrow count$

Implementation



A dynamic instrumentation infrastructure that translates the binary code into an architecture-neutral intermediate representation (`VEX`)

Events	Instrumentation	Data structures
memory accesses	easy	shadow memory
threads	easy	thread state
function calls/returns	hard	shadow stack

SPEC CPU2006 – Time (slowdown)

	memcheck	callgrind-base	callgrind-cache	aprof
CINT	15.7×	46.5×	98.8×	31.8×
CFP	21.3×	20.4×	92.7×	27.9×

`memcheck` does not trace function calls/returns

`callgrind-base` does not trace memory accesses

SPEC CPU2006 – Time (slowdown)

	memcheck	callgrind-base	callgrind-cache	aprof
CINT	15.7×	46.5×	98.8×	31.8×
CFP	21.3×	20.4×	92.7×	27.9×

`memcheck` does not trace function calls/returns
`callgrind-base` does not trace memory accesses

$\Longrightarrow$ `aprof` delivers comparable perfomance wrt other Valgrind tools

SPEC CPU2006 – Time (slowdown)

	memcheck	callgrind-base	callgrind-cache	aprof
CINT	15.7×	46.5×	98.8×	31.8×
CFP	21.3×	20.4×	92.7×	27.9×

`memcheck` does not trace function calls/returns

`callgrind-base` does not trace memory accesses

$\Longrightarrow$ `aprof` delivers comparable perfomance wrt other Valgrind tools

$\Longrightarrow$ a chart with k points:

- 1 run with `aprof`
- k runs with `gprof`

SPEC CPU2006 – Space (overhead)

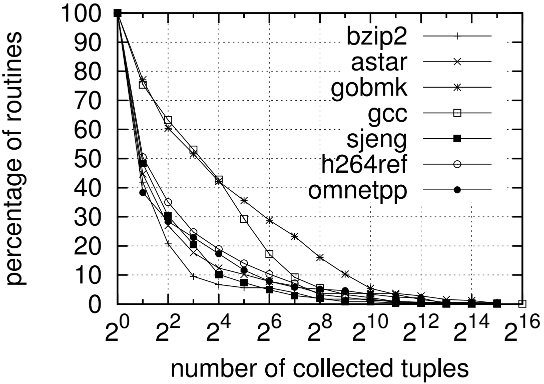
	`memcheck`	`callgrind-base`	`callgrind-cache`	`aprof`
CINT	1.8×	1.3×	1.3×	2.2×
CFP	1.5×	1.3×	1.3×	1.9×

`callgrind-base` does not use a shadow memory

`memcheck` applies different compression schemes

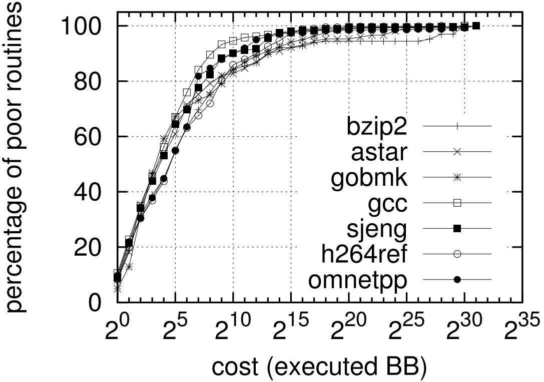
How many perfomance tuples?

How many performance tuples can be automatically collected for each routine from a ***single run*** of a program on a typical workload?

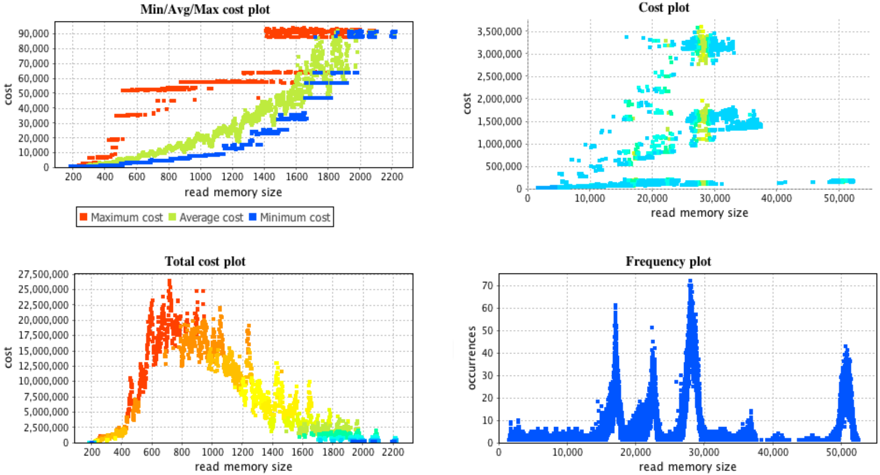


Are “poor” routines interesting?

“poor” routines = routines with less than 10 tuples

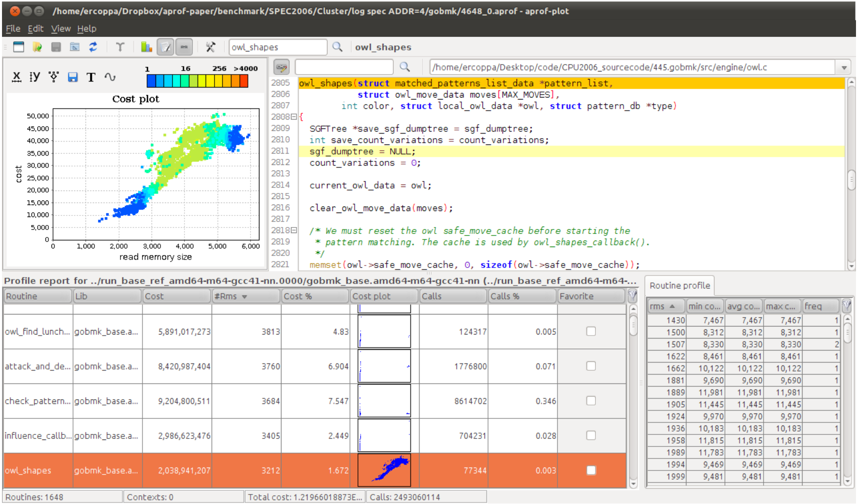


Some profiles by aprof for SPEC CPU2006 benchmarks



More charts at the poster session!

aprof-plot: interactive graphical viewer for aprof profiles



Thanks!

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