Input-Sensitive Profiling

Emilio Coppa
Camil Demetrescu
Irene Finocchi

June 11, 2012

PLDI 2012
Conventional profilers gather **cumulative info** over a whole execution.

\[ f_a(n) \Rightarrow \]

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:

\[ \langle n_i, c_i, \text{max}_i, \text{min}_i, \text{sum}_i, q_i \rangle \]

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 \( n_i \)
- \( \text{max}_i/\text{min}_i \) = maximum and minimum costs required by any execution of f on input size \( n_i \)
- \( \text{sum}_i/q_i \) = 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.
void swap(int * a, int * b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

The function \texttt{swap} has RMS 2 because it reads (first access) objects \texttt{*a} and \texttt{*b}, and writes (first access) variable \texttt{temp}.
### Read Memory Size (Example 2)

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

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
```

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>
Read Memory Size (Example 2)

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

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>[x] y</td>
<td>1</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>x, y</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>x y</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>
call f
  read x
  write y
call g
  read x
  read y
  read z
  write w
return
read w
return

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>(x) (y)</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>(x) (y)</td>
<td>2</td>
</tr>
</tbody>
</table>
Intro RMS Case study Alg. Implem. Experiments
Our approach Definition Examples

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

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>![Accessed cells](x y z)</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>![Accessed cells](x y z)</td>
<td>3</td>
</tr>
</tbody>
</table>

7 / 23 E. Coppa, C. Demetrescu, I. Finocchi

Input-Sensitive Profiling, PLDI 2012
Read Memory Size (Example 2)

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

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>x, y, z, w</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>x, y, z, w</td>
<td>3</td>
</tr>
</tbody>
</table>
call f
read x
write y
call g
read x
read y
read z
write w
return
read w
return

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>x y z w</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>x y z w</td>
<td>3</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>x, y, z, w</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>x, y, z, w</td>
<td>3</td>
</tr>
</tbody>
</table>
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
```

<table>
<thead>
<tr>
<th>Fn</th>
<th>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>x y z w</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>x y z w</td>
<td>3</td>
</tr>
</tbody>
</table>

7/23 E. Coppa, C. Demetrescu, I. Finocchi

Input-Sensitive Profiling, PLDI 2012
We discuss \texttt{wf-0.41}, a simple word frequency counter included in the current development head of Linux Fedora (Fedora 17–Beefy Miracle).

We profile \texttt{wf} with:

- \texttt{gprof} a traditional and well-known call graph execution profiler – http://www.gnu.org/software/binutils/
- \texttt{apropf} asymptotic profiler
  our implementation of an input-sensitive profiler – http://code.google.com/p/apropf/
We discuss \texttt{wf-0.41}: \texttt{gprof} vs \texttt{aprof}
We discuss \texttt{wf-0.41}: gprof vs aprof
Is there any bottleneck in str_tolower?

```c
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?
Is there any bottleneck in \texttt{str\_tolower}?

\begin{verbatim}
void str_tolower(char* str) {
    int i;
    for (i = 0; i < strlen(str); i++)
        str[i] = wf_tolower(str[i]);
}
\end{verbatim}
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?
Input of `str_tolower` = single words of input text **not** the input text!

---

E. Coppa, C. Demetrescu, I. Finocchi

Input-Sensitive Profiling, PLDI 2012
Short vs long words in gprof

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

<table>
<thead>
<tr>
<th>Input: Anna Karenina</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>52.2%</td>
<td>addword</td>
</tr>
<tr>
<td></td>
<td>31.3%</td>
<td><code>str_tolower</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input: Protein sequences</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>61.8%</td>
<td><code>str_tolower</code></td>
</tr>
<tr>
<td></td>
<td>32.6%</td>
<td>addword</td>
</tr>
</tbody>
</table>
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:

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

Improvements:

<table>
<thead>
<tr>
<th>Anna Karenina</th>
<th>6%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protein sequences</td>
<td>30%</td>
</tr>
</tbody>
</table>
Computing RMS: data structures

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

- $\textit{rtn}$: id of the routine
- $\textit{ts}$: timestamp assigned to this activation
- $\textit{cost}$: cumulative cost
- $\textit{rms}$: 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)

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

<table>
<thead>
<tr>
<th>Fn</th>
<th>RMS</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>f</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Step 0: count = 0

TIMESTAMPS

ts[x] = 0
ts[y] = 0
ts[z] = 0
ts[w] = 0
Computing RMS (example)

**Step 1:** \( \text{count} = 1 \)

<table>
<thead>
<tr>
<th>Fn</th>
<th>RMS</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>f</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

**SHADOW RUNTIME STACK**

- \( S[0].id = f \)
- \( S[0].cost = 1 \)
- \( S[0].ts = 1 \)
- \( S[0].rms = 0 \)

**TIMESTAMPS**

- \( ts[x] = 0 \)
- \( ts[y] = 0 \)
- \( ts[z] = 0 \)
- \( ts[w] = 0 \)

---

**Intro RMS Case study Alg. Implement. Experiments**

**Computing RMS (example)**

- **call f**
  - read x
  - write y
- **call g**
  - read x
  - read y
  - read z
  - write w
  - return
- read w
- return
Computing RMS (example)

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

<table>
<thead>
<tr>
<th>Fn</th>
<th>RMS</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>f</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Step 2: \( \text{count} = 1 \)

**SHADOW RUNTIME STACK**

- \( S[0].id = f \)
- \( S[0].cost = 1 \)
- \( S[0].ts = 1 \)
- \( S[0].rms = 1 \)

**TIMESTAMPS**

- \( ts[x] = 1 \)
- \( ts[y] = 0 \)
- \( ts[z] = 0 \)
- \( ts[w] = 0 \)
Computing RMS (example)

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

Step 3: count = 1

SHADOW RUNTIME STACK

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

TIMESTAMP

ts[x] = 1

Ts[y] = 1

Ts[z] = 0

Ts[w] = 0

<table>
<thead>
<tr>
<th>Fn</th>
<th>RMS</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>f</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Computing RMS (example)

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

<table>
<thead>
<tr>
<th>Fn</th>
<th>RMS</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>f</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Step 4: count = 2

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)

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

<table>
<thead>
<tr>
<th>Fn</th>
<th>RMS</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>f</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Step 5: count = 2

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)

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

<table>
<thead>
<tr>
<th>Fn</th>
<th>RMS</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g$</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>$f$</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Step 6: count = 2

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
nts[$y$] = 2
nts[$z$] = 2
nts[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

<table>
<thead>
<tr>
<th>Fn</th>
<th>RMS</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>f</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Step 7: \( \text{count} = 2 \)

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

```
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

<table>
<thead>
<tr>
<th>Fn</th>
<th>RMS</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>f</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Step 8: \( \text{count} = 2 \)

**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

Step 9:  count = 2

<table>
<thead>
<tr>
<th>Fn</th>
<th>RMS</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>f</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

ts\[x\] = 2
ts\[y\] = 2
ts\[z\] = 2
ts\[w\] = 2
Computing RMS: algorithm

**procedure** call(r): $O(1)$

- $top++$
- $S[top].rtn \leftarrow r$
- $S[top].ts \leftarrow ++count$
- $S[top].rms \leftarrow 0$
- $S[top].cost \leftarrow \text{get\_cost()}$

**procedure** return(): $O(1)$

- collect($S[top].rtn$, $S[top].rms$
- \text{get\_cost()} - $S[top].cost$
- $S[top-1].rms += S[top].rms$
- $top--$

**procedure** read(w): $O(\log(\text{stack\ depth}))$

- 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): $O(1)$

- $ts[w] \leftarrow count$
A dynamic instrumentation infrastructure that translates the binary code into an architecture-neutral intermediate representation (**VEX**)

<table>
<thead>
<tr>
<th>Events</th>
<th>Instrumentation</th>
<th>Data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory accesses</td>
<td>easy</td>
<td>shadow memory</td>
</tr>
<tr>
<td>threads</td>
<td>easy</td>
<td>thread state</td>
</tr>
<tr>
<td>function calls/returns</td>
<td>hard</td>
<td>shadow stack</td>
</tr>
</tbody>
</table>
memcheck does not trace function calls/returns

callgrind-base does not trace memory accesses
### SPEC CPU2006 – Time (slowdown)

<table>
<thead>
<tr>
<th></th>
<th>memcheck</th>
<th>callgrind-base</th>
<th>callgrind-cache</th>
<th>aprof</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINT</td>
<td>15.7×</td>
<td>46.5×</td>
<td>98.8×</td>
<td>31.8×</td>
</tr>
<tr>
<td>CFP</td>
<td>21.3×</td>
<td>20.4×</td>
<td>92.7×</td>
<td>27.9×</td>
</tr>
</tbody>
</table>

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

⇒ **aprop** delivers comparable performance wrt other Valgrind tools
memcheck does not trace function calls/returns

callgrind-base does not trace memory accesses

⇒ aprof delivers comparable performance wrt other Valgrind tools

⇒ a chart with k points:
  - 1 run with aprof
  - k runs with gprof
SPEC CPU2006 – Space (overhead)

<table>
<thead>
<tr>
<th></th>
<th>memcheck</th>
<th>callgrind-base</th>
<th>callgrind-cache</th>
<th>aprof</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINT</td>
<td>1.8×</td>
<td>1.3×</td>
<td>1.3×</td>
<td>2.2×</td>
</tr>
<tr>
<td>CFP</td>
<td>1.5×</td>
<td>1.3×</td>
<td>1.3×</td>
<td>1.9×</td>
</tr>
</tbody>
</table>

callgrind-base does not use a shadow memory  
memcheck applies different compression schemes
How many performance tuples?

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

![Graph showing percentage of routines against number of collected tuples for various programs.

- bzip2
- astar
- gobmk
- gcc
- sjeng
- h264ref
- omnetpp

The graph illustrates the percentage of routines against the number of collected tuples for different programs, showing how the percentage decreases as the number of collected tuples increases.](image-url)
Are “poor” routines interesting?

“poor” routines = routines with less than 10 tuples

![Graph showing the percentage of poor routines vs. cost (executed BB) for various programs. The x-axis represents the cost (executed BB), ranging from $2^0$ to $2^{35}$. The y-axis represents the percentage of poor routines, ranging from 0 to 100. Different line styles and markers are used to represent different programs: bzip2, astar, gobmk, gcc, sjeng, h264ref, and omnetpp. The graph indicates a trend where the percentage of poor routines increases with cost for all programs, with some programs showing a sharper increase than others.]
Some profiles by aprof for SPEC CPU2006 benchmarks

More charts at the poster session!
aprop-plot: interactive graphical viewer for aprof profiles
Download aprof at:
http://code.google.com/p/aprof/