Estimating the Empirical Cost Function of Routines with Dynamic Workloads

Emilio Coppa
Camil Demetrescu
Irene Finocchi
Romolo Marotta

February 18, 2014
Performance Scalability Analysis

Theory:
Asymptotic Analysis

- Mergesort: \( T(n) = \Theta(n \log n) \)
- Bubble sort: \( T(n) = \Theta(n^2) \)

Our goal:
Predicting how code scales w.r.t. its workload size.
Performance Scalability Analysis

Theory:
Asymptotic Analysis

Practice:
Performance Profiling

<table>
<thead>
<tr>
<th>%time</th>
<th>%self</th>
<th>%children</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.9</td>
<td>2.00</td>
<td>16.29</td>
<td>main</td>
</tr>
<tr>
<td>52.2</td>
<td>5.82</td>
<td>3.70</td>
<td>foo</td>
</tr>
<tr>
<td>31.3</td>
<td>2.81</td>
<td>2.90</td>
<td>bar</td>
</tr>
<tr>
<td>20.3</td>
<td>3.70</td>
<td>0.00</td>
<td>foobar</td>
</tr>
</tbody>
</table>
Performance Scalability Analysis

Theory:
Asymptotic Analysis

- **mergesort**
  
  \[ T(n) = O(n \log n) \]

- **bubblesort**
  
  \[ T(n) = O(n^2) \]

Practice:
Performance Profiling

<table>
<thead>
<tr>
<th>%time</th>
<th>%self</th>
<th>%children</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.9</td>
<td>2.00</td>
<td>16.29</td>
<td>main</td>
</tr>
<tr>
<td>52.2</td>
<td>5.82</td>
<td>3.70</td>
<td>foo</td>
</tr>
<tr>
<td>31.3</td>
<td>2.81</td>
<td>2.90</td>
<td>bar</td>
</tr>
<tr>
<td>20.3</td>
<td>3.70</td>
<td>0.00</td>
<td>foobar</td>
</tr>
</tbody>
</table>

Our goal:
Predicting how code scales w.r.t. its workload size
Analyzing the scalability of routines

A possible solution:

1. Choose workloads of increasing sizes

1 MB
5 MB
9 MB
...

Coppa, Demetrescu, Finocchi, Marotta
Analyzing the scalability of routines

A possible solution:

1. Choose workloads of increasing sizes
2. For each workload: run your application under `gprof`

---

1 MB
5 MB
9 MB

`gprof`
`gmon.out`
`gprof`
`gmon.out`
`gprof`
`gmon.out`
Analyzing the scalability of routines

A possible solution:

1. Choose workloads of increasing sizes
2. For each workload: run your application under `gprof`
3. Plot and analyze the results
Drawbacks of gprof-like experiments

Need to have different workloads for different routines:
application’s workload ≠ routine’s workloads

Wrong assumptions can easily lead to misleading conclusions:

\[
\text{str}_\text{tolower} \quad \text{versus} \quad \text{str}_\text{tolower}
\]

See case studies in [CDF12]
Workload-dependent profiling

Recent works investigate how an application’s performance scales as a function of its input data:

Main sources of dynamic workloads

<table>
<thead>
<tr>
<th>Input Size Estimation</th>
<th>I/O &amp; Syscall</th>
<th>Thread Intercom.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAW07</td>
<td>Manual</td>
<td></td>
</tr>
<tr>
<td>ZH12</td>
<td>Automatic</td>
<td>No</td>
</tr>
<tr>
<td>CDF12</td>
<td>Automatic</td>
<td>No</td>
</tr>
</tbody>
</table>

GAW07 Goldsmith, Aiken, and Wilkerson, Measuring empirical computational complexity, ESEC/FSE 2007

ZH12 Zaparanuks and Hauswirth, Algorithmic profiling, PLDI 2012

CDF12 Coppa, Demetrescu, and Finocchi, Input-Sensitive Profiling, PLDI 2012
Dynamic workloads are ubiquitous!

Many routines dynamically receive input values during their activations:

- Thread intercommunications: e.g., producer-consumer pattern
- I/O operations via syscalls: e.g., buffered read operations

If ignoring dynamic workloads, then input size estimation may be wrong

Analysis of profiling data may be misleading
Our contribution

Main sources of dynamic workloads

<table>
<thead>
<tr>
<th></th>
<th>Input Size Estimation</th>
<th>I/O &amp; Syscall</th>
<th>Thread Intercom.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GAW07</strong></td>
<td>Manual</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ZH12</strong></td>
<td>Automatic</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>CDF12</strong></td>
<td>Automatic</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>CDFM14</strong></td>
<td>Automatic</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**GAW07** Goldsmith, Aiken, and Wilkerson, Measuring empirical computational complexity, ESEC/FSE 2007

**ZH12** Zaparanuks and Hauswirth, Algorithmic profiling, PLDI 2012

**CDF12** Coppa, Demetrescu, and Finocchi, Input-Sensitive Profiling, PLDI 2012
Estimating the input size: previous approaches

[ZH12] Input size $\approx$ Size of (Java) Data Structures

Input size definition depends on the specific data structure (e.g., if array, then array size). Not suitable for low-level programming languages.
Estimating the input size: previous approaches

[ZH12] Input size \( \approx \) Size of (Java) Data Structures

Input size definition depends on the specific data structure (e.g., if array, then array size). Not suitable for low-level programming languages.

[CDF12] Input size \( \approx \) Read memory size (RMS)

Read memory size of an execution of a routine \( r \)

\[
= \text{number of distinct memory cells first accessed by } r \text{ (or by a descendant of } r \text{ in the call tree) with a read operation}
\]

This work extends the RMS metric
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>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RMS example

<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>

call f
read x
write y
call g
read x
read y
read z
write w
return
read w
return
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>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>
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>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>
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>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>( \text{\color{green}x}, y )</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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>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>
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>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>
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>Accessed cells (first-read green)</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>x y z</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>x y z</td>
<td>3</td>
</tr>
</tbody>
</table>
### RMS Example

```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, z, w</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>x, y, z, w</td>
<td>3</td>
</tr>
</tbody>
</table>
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>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>

Input-Sensitive Profiling with Dynamic Workloads - CGO'14
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>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>
RMS example

```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, z, w</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>x, y, z, w</td>
<td>3</td>
</tr>
</tbody>
</table>
When RMS does not work: an example

Multithreading may be an issue:

gets a value produced by another thread

RMS\(_f\) = 1, but actual input size is 2!

RMS fails to properly characterize the input size of routine activations under dynamic workloads
From RMS to Dynamic Read Memory Size (DRMS)

\[ r = \text{routine activation} \]
\[ t = \text{thread} \]
\[ \ell = \text{memory location} \]

A read operation on \( \ell \) is:

**First-read**
if \( \ell \) has never been accessed before by \( r \) or by any of its descendants in the call tree of thread \( t \).

**Induced first-read**
if no previous access to \( \ell \) has been made by \( t \) since the latest write to \( \ell \) performed by a thread different from \( t \), if any.
$T_1$ did not access location $x$ since the latest write to $x$ performed by $T_2$.

The second read $x$ is an induced first-read.
Dynamic Read Memory Size (DRMS)

Input size $\approx$ Dynamic Read Memory Size

$$\text{DRMS}_{r,t} := \# \text{ of first-reads or } \text{induced first-reads}$$

Notice that:

$$\text{RMS}_{r,t} := \# \text{ of first-reads}$$
DRMS example

First read x: first-read
Second read x: induced first-read

DRMS_{f,T_1} = 2

These kinds of patterns occur frequently in real applications
Pattern 1: producer-consumer

**Procedure producer()**

1: while (1) do
2: wait(empty)
3: wait(mutex)
4: \(x = \text{produceData}()\)
5: signal(mutex)
6: signal(full)

**Procedure consumer()**

1: while (1) do
2: wait(full)
3: wait(mutex)
4: \(\text{consumeData}(x)\)
5: signal(mutex)
6: signal(empty)

always the same memory location across iterations!

When producer has generated \(n\) values:

\[
\text{RMS}_{\text{consumer}} = 1 \quad \text{while} \quad \text{DRMS}_{\text{consumer}} = n
\]
Pattern 2: data streaming

```plaintext
procedure streamReader()
  1:  for i = 1 to n do
  2:    fill x with external data from the network
  3:    consumeData (x)

always the same memory location across iterations!
```

At the end of the execution:

\[
\text{RMS}_{\text{streamReader}, t} = 1 \quad \text{while} \quad \text{DRMS}_{\text{streamReader}, t} = n
\]
Computing DRMS: a simple-minded approach

\[ r = \text{routine activation} \quad t = \text{thread} \]
\[ l = \text{memory location} \quad L_{r,t} = \text{set of locations accessed by } r \]
Computing DRMS: a simple-minded approach

\( r = \) routine activation \hspace{1cm} \( t = \) thread
\( \ell = \) memory location \hspace{1cm} \( L_{r,t} = \) set of locations accessed by \( r \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
</table>
| \( \text{read}_t(\ell) \) | \( \ell \notin L_{r,t} \) | \( \text{DRMS}_{r,t} \)++  
\( L_{r,t} \leftarrow L_{r,t} \cup \{\ell\} \) |
| \( \text{write}_t(\ell) \) | | \( L_{r,t} \leftarrow L_{r,t} \cup \{\ell\} \) |
| \( \text{write}_{t'}(\ell), t' \neq t \) | | \( L_{r,t} \leftarrow L_{r,t} \setminus \{\ell\} \) |
Computing DRMS: a simple-minded approach

$\mathbf{r} =$ routine activation \\
$\mathbf{t} =$ thread \\
$\mathbf{\ell} =$ memory location \\
$\mathbf{L}_{r,t} =$ set of locations accessed by $\mathbf{r}$

<table>
<thead>
<tr>
<th>Operation</th>
<th>Condition</th>
<th>DRMS Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{read}_t(\ell)$</td>
<td>$\ell \notin \mathbf{L}_{r,t}$</td>
<td>$\mathbf{DRMS}<em>{r,t} \mathcal{+}$ \hspace{1cm} $\mathbf{L}</em>{r,t} \leftarrow \mathbf{L}_{r,t} \cup {\ell}$</td>
</tr>
<tr>
<td>$\text{write}_t(\ell)$</td>
<td></td>
<td>$\mathbf{L}<em>{r,t} \leftarrow \mathbf{L}</em>{r,t} \cup {\ell}$</td>
</tr>
<tr>
<td>$\text{write}_{t'}(\ell), t' \neq t$</td>
<td></td>
<td>$\mathbf{L}<em>{r,t} \leftarrow \mathbf{L}</em>{r,t} \setminus {\ell}$</td>
</tr>
</tbody>
</table>

Repeat for all pending rtn activations in the call stack
Computing DRMS: a simple-minded approach

\( r = \) routine activation \hspace{1cm} \( t = \) thread
\( \ell = \) memory location \hspace{1cm} \( L_{r,t} = \) set of locations accessed by \( r \)

| read\(_t\)(\(\ell\)) | if \( \ell \notin L_{r,t} \) then \( \text{DRMS}_{r,t}++ \)
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( L_{r,t} \leftarrow L_{r,t} \cup {\ell} )</td>
</tr>
<tr>
<td>write(_t)((\ell))</td>
<td>( L_{r,t} \leftarrow L_{r,t} \cup {\ell} )</td>
</tr>
<tr>
<td>write(_{t'})((\ell)), ( t' \neq t )</td>
<td>( L_{r,t} \leftarrow L_{r,t} \setminus {\ell} )</td>
</tr>
</tbody>
</table>

- Repeat for all pending rtn activations in the call stack
- Repeat for all rtn activations in any stack
Computing DRMS: a simple-minded approach

\[ r = \text{routine activation} \quad t = \text{thread} \]
\[ \ell = \text{memory location} \quad L_{r,t} = \text{set of locations accessed by } r \]

<table>
<thead>
<tr>
<th>operation</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{read}_t(\ell) )</td>
<td>if ( \ell \notin L_{r,t} ) then ( \text{DRMS}<em>{r,t}++ ) ( L</em>{r,t} \leftarrow L_{r,t} \cup {\ell} )</td>
</tr>
<tr>
<td>( \text{write}_t(\ell) )</td>
<td>( L_{r,t} \leftarrow L_{r,t} \cup {\ell} )</td>
</tr>
<tr>
<td>( \text{write}_{t'}(\ell), \ t' \neq t )</td>
<td>( L_{r,t} \leftarrow L_{r,t} \setminus {\ell} )</td>
</tr>
</tbody>
</table>

- Repeat for all pending rtn activations in the call stack
- Repeat for all rtn activations in any stack (on a write)

\[ O \left( \sum_{t \in \text{Threads}} |\text{Stack}_t| \right) \quad \text{time per access} \]

\[ O \left( \sum_{t \in \text{Threads}} \cdot \sum_{r \in \text{Stack}_t} |L_{r,t}| \right) \quad \text{current memory footprint} \]
Computing DRMS efficiently

Our solution based on:
- a timestamp algorithm
- a global shadow memory
- thread-private shadow memory for each thread
- periodic global renumbering algorithm

\[
O\left(\log |S_{current\,Thread}|\right) \quad \text{time per access:}
\]
\[
O\left(\sum_{t \in Threads} |\text{accessed locations by } t|\right) \quad \text{memory footprint:}
\]

Details in the paper
aprof-drms is based on the Valgrind framework, 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>
<tr>
<td>system calls</td>
<td>easy</td>
<td>shadow memory</td>
</tr>
</tbody>
</table>
A case study on MySQL

Repeating a select query on tables of increasing sizes:

RMS may be misleading with I/O bound or multithreaded applications!
A case study on vips

Processing some images of increasing sizes:

DRMS typically yields richer profiles than RMS
Routine-by-routine thread and external input

MySQL

% of dynamic workload

Thread input
External input

% routines (sorted by amount of dynamic workload)

67%

vips

% of dynamic workload

Thread input
External input

% routines (sorted by amount of dynamic workload)

70%
Characterization of induced first-reads

SPEC OMP2012

PARSEC 2.1

% thread / external input

nab kdtree botsspar imagick smithwa swaptions botsalign fluidanimate canneal dedup bodytrack ferret x264 blackscholes

Thread input  External input
aprof-plot: interactive graphical viewer for aprof profiles
Download aprof at:
http://code.google.com/p/aprof/
Both benchmark suites were set up for running 4 threads:

<table>
<thead>
<tr>
<th></th>
<th>memcheck</th>
<th>helgrind</th>
<th>aprof</th>
<th>aprof-drms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Slowdown (Geom. Mean)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPEC OMP</td>
<td>94.1×</td>
<td>179.4×</td>
<td>101.5×</td>
<td>140.8×</td>
</tr>
<tr>
<td>PARSEC 2.1</td>
<td>51.8×</td>
<td>153.3×</td>
<td>57.1×</td>
<td>68.2×</td>
</tr>
<tr>
<td><strong>Space overhead (Geom. Mean)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPEC OMP</td>
<td>2.0×</td>
<td>4.5×</td>
<td>2.8×</td>
<td>3.3×</td>
</tr>
<tr>
<td>PARSEC 2.1</td>
<td>2.9×</td>
<td>8.4×</td>
<td>4.6×</td>
<td>6.1×</td>
</tr>
</tbody>
</table>

- All tools suffer Valgrind serialization
- aprof-drms delivers comparable performance wrt other Valgrind tools