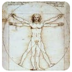
SAPIENZA
UNIVERSITÀ DI ROMA

Estimating the Empirical Cost Function of Routines with Dynamic Workloads

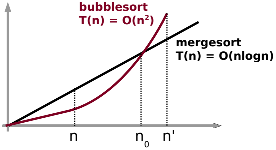
Emilio Coppa
Camil Demetrescu
Irene Finocchi
Romolo Marotta

February 18, 2014

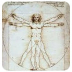
Performance Scalability Analysis



Theory:
Asymptotic Analysis

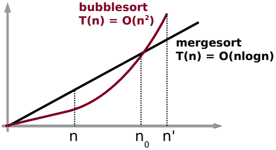


Performance Scalability Analysis



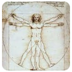
Theory:
Asymptotic Analysis

Practice:
Performance Profiling



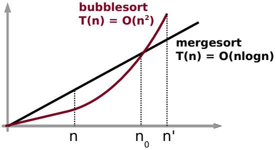
%time	%self	%children	function
99.9	2.00	16.29	`main`
52.2	5.82	3.70	`foo`
31.3	2.81	2.90	`bar`
20.3	3.70	0.00	`foobar`

Performance Scalability Analysis

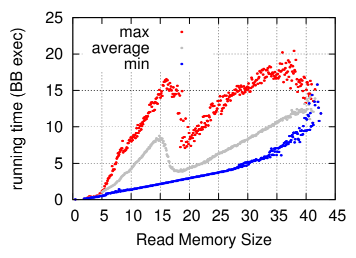


Theory:
Asymptotic Analysis

Practice:
Performance Profiling



%time	%self	%children	function
99.9	2.00	16.29	main
52.2	5.82	3.70	foo
31.3	2.81	2.90	bar
20.3	3.70	0.00	foobar

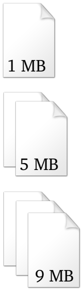


Our goal:
Predicting how code scales w.r.t. its workload size

Analyzing the scalability of routines

A possible solution:

1. Choose workloads of increasing sizes

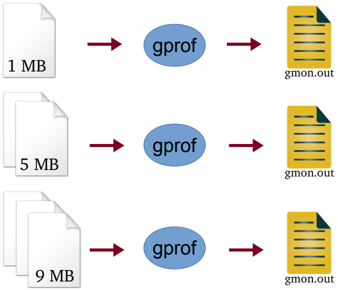


....

Analyzing the scalability of routines

A possible solution:

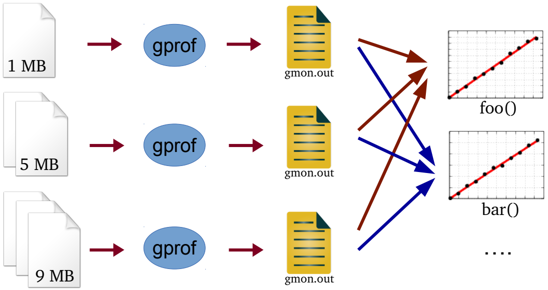
1. Choose workloads of increasing sizes
2. For each workload: run your application under `gprof`



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

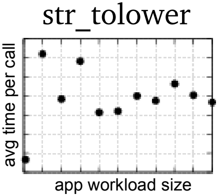


Drawbacks of gprof-like experiments

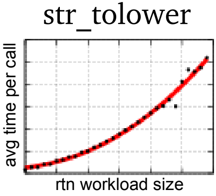
Need to have different workloads for different routines:

application's workload $\neq$ routine's workloads

Wrong assumptions can easily lead to misleading conclusions:



versus



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

	Input Size Estimation	I/O & Syscall	Thread Intercom.
GAW07	Manual		
ZH12	Automatic	No	No
CDF12	Automatic	No	No

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

$$\overbrace{\hphantom{\text{I/O \& Syscall Thread Intercom.}}}^{\text{Main sources of dynamic workloads}}$$

	Input Size Estimation	I/O & Syscall	Thread Intercom.
GAW07	Manual		
ZH12	Automatic	No	No
CDF12	Automatic	No	No
CDFM14	Automatic	Yes	Yes

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 ≈ 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 ≈ **Read memory size** (RMS)

Read memory size of an execution of a routine r

=

number of distinct memory cells
first accessed by r (or by a descendant of r 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
```

Fn	Accessed cells (first-read green)	RMS

RMS example

```
→ 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		

RMS example

```
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

RMS example

```
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

RMS example

```
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		

RMS example

```
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

RMS example

```
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

RMS example

```
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

RMS example

```
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

RMS example

```
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

RMS example

```
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

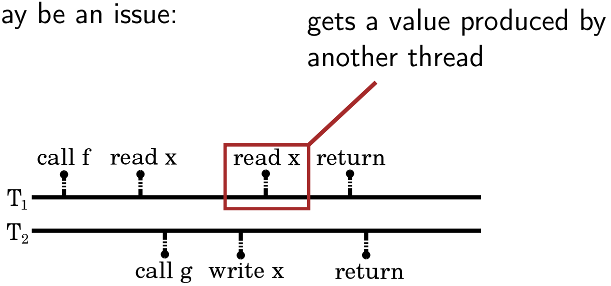
RMS example

```
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

When RMS does not work: an example

Multithreading may be an issue:



$\mathrm{RMS}_f = 1$, 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 = routine activation
t = thread
ℓ = memory location

A read operation on ℓ is:

First-read

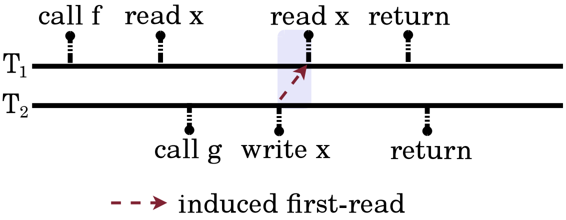
if ℓ has never been accessed before by r or by any of its descendants in the call tree of thread t.

Induced first-read

if no previous access to ℓ has been made by t since the latest write to ℓ performed by a thread different from t, if any.

Induced first-read example

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)

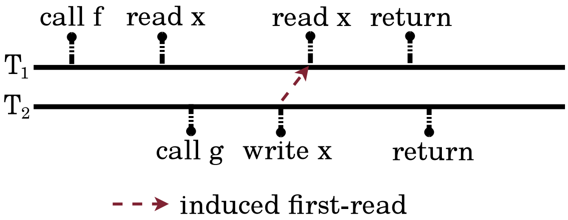
Input size $\approx$ Dynamic Read Memory Size

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

Notice that:

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

DRMS example



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

$\text{DRMS}_{f,T_1} = 2$

These kinds of patterns occur frequently in real applications

Pattern 1: producer-consumer

procedure `producer()`

while (1) **do**
    `wait(`*empty*`)`
    `wait(`*mutex*`)`
    x = `produceData()`
    `signal(`*mutex*`)`
    `signal(`*full*`)`

procedure `consumer()`

while (1) **do**
    `wait(`*full*`)`
    `wait(`*mutex*`)`
    `consumeData` $\boxed{(x)}$
    `signal(`*mutex*`)`
    `signal(`*empty*`)`

always the same memory location across iterations!

When `producer` has generated n values:

$\mathrm{RMS}_{consumer} = 1$ while $\mathrm{DRMS}_{consumer} = n$

Pattern 2: data streaming

procedure `streamReader()`

for $i = 1$ to n **do**

    fill x with external data from the network

    `consumeData` $\boxed{(x)}$

always the same memory location across iterations!

At the end of the execution:

$\mathrm{RMS}_{\texttt{streamReader},t} = 1$ while $\mathrm{DRMS}_{\texttt{streamReader},t} = n$

Computing DRMS: a simple-minded approach

$r =$ routine activation

$\ell =$ memory location

$t =$ thread

$L_{r,t} =$ set of locations accessed by r

Computing DRMS: a simple-minded approach

$r =$ routine activation

$\ell =$ memory location

$t =$ thread

$L_{r,t} =$ set of locations accessed by r

$\texttt{read}_t(\ell)$	if $\ell \notin L_{r,t}$ then $\mathrm{DRMS}_{r,t}$++ $L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\texttt{write}_t(\ell)$	$L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\texttt{write}_{t'}(\ell)$, $t' \neq t$	$L_{r,t} \leftarrow L_{r,t} \setminus \{\ell\}$

Computing DRMS: a simple-minded approach

r = routine activation

ℓ = memory location

t = thread

$L_{r,t}$ = set of locations accessed by r

$\mathtt{read}_t(\ell)$	if $\ell \notin L_{r,t}$ then $\mathrm{DRMS}_{r,t}$++ $L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\mathtt{write}_t(\ell)$	$L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\mathtt{write}_{t'}(\ell),\ t' \neq t$	$L_{r,t} \leftarrow L_{r,t} \setminus \{\ell\}$

- Repeat for all pending rtn activations in the call stack

Computing DRMS: a simple-minded approach

r = routine activation

ℓ = memory location

t = thread

$L_{r,t}$ = set of locations accessed by r

$\texttt{read}_t(\ell)$	if $\ell \notin L_{r,t}$ then $\mathrm{DRMS}_{r,t}$++ $L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\texttt{write}_t(\ell)$	$L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\texttt{write}_{t'}(\ell)$, $t' \neq t$	$L_{r,t} \leftarrow L_{r,t} \setminus \{\ell\}$

- 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 = routine activation

ℓ = memory location

t = thread

$L_{r,t}$ = set of locations accessed by r

$\texttt{read}_t(\ell)$	if $\ell \notin L_{r,t}$ then $\text{DRMS}_{r,t}$++ $L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\texttt{write}_t(\ell)$	$L_{r,t} \leftarrow L_{r,t} \cup \{\ell\}$
$\texttt{write}_{t'}(\ell)$, $t' \neq t$	$L_{r,t} \leftarrow L_{r,t} \setminus \{\ell\}$

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

$$O\left(\sum_{t \in Threads} |Stack_t|\right)$$

time per access

$$O\left(\sum_{t \in Threads} \cdot \sum_{r \in Stack_t} |L_{r,t}|\right)$$

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

time per access:

$$O\left(\log |S_{currentThread}|\right)$$

memory footprint:

$$O\left(\sum_{t \in Threads} |\text{accessed locations by } t|\right)$$

Details in the paper

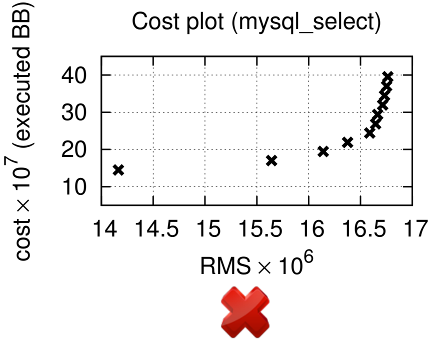
Implementation

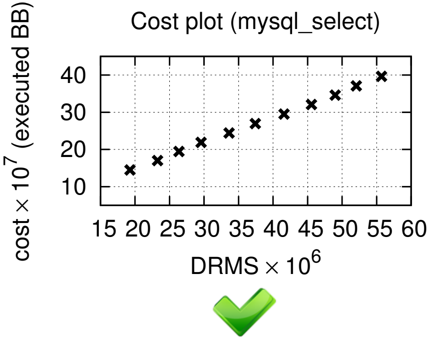
`aprof-drms` is based on the `Valgrind` framework, a dynamic instrumentation infrastructure that translates the binary code into an architecture-neutral intermediate representation (`VEX`)

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

A case study on `MySQL`

Repeating a select query on tables of increasing sizes:

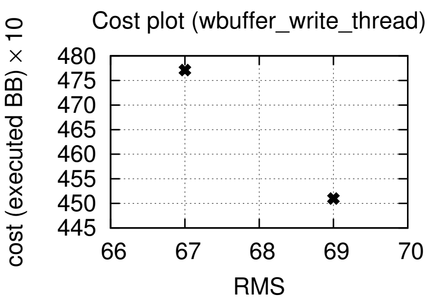


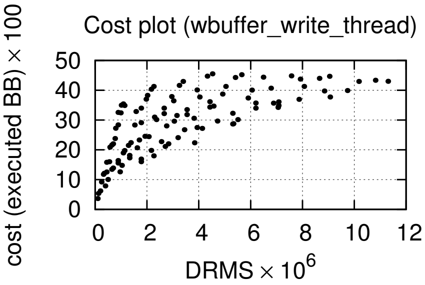


RMS may be misleading with I/O bound or multithreaded applications!

A case study on vips

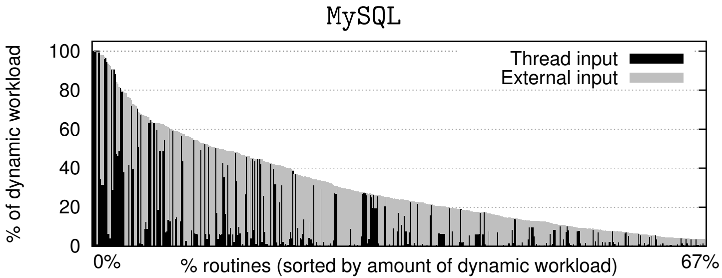
Processing some images of increasing sizes:

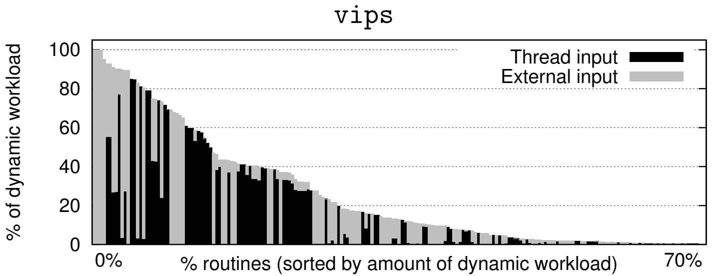




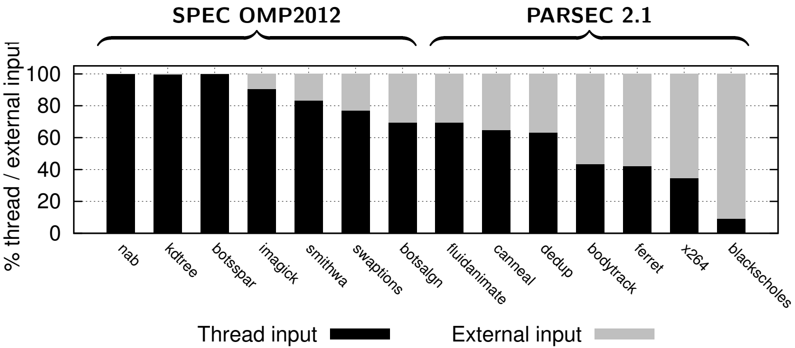
DRMS tipically yields richer profiles than RMS

Routine-by-routine thread and external input



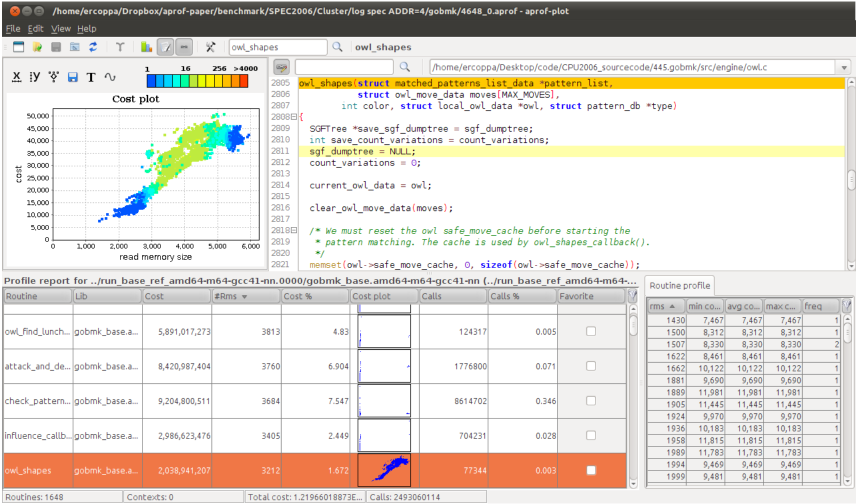


Characterization of induced first-reads



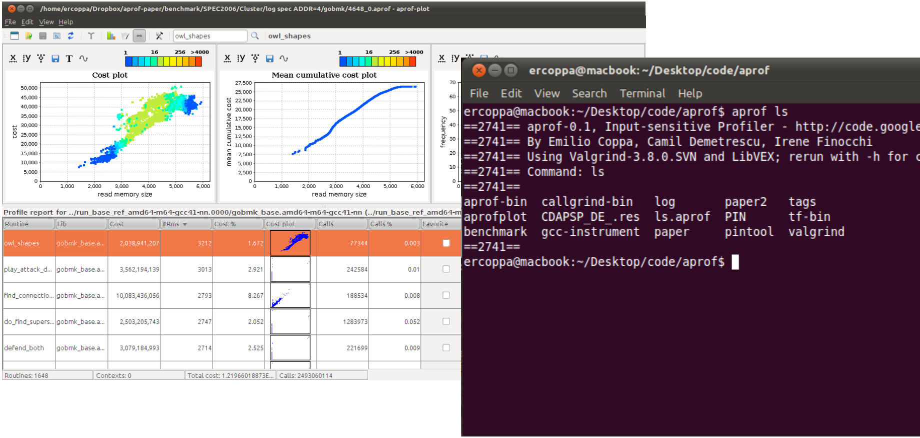
Advertisement ☺

aprof-plot: interactive graphical viewer for aprof profiles



Thanks!

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



Slowdown & space overhead

Both benchmark suites were set up for running 4 threads:

	`memcheck`	`helgrind`	`aprof`	`aprof-drms`
Slowdown (Geom. Mean)				
SPEC OMP	94.1×	179.4×	101.5×	140.8×
PARSEC 2.1	51.8×	153.3×	57.1×	68.2×
Space overhead (Geom. Mean)				
SPEC OMP	2.0×	4.5×	2.8×	3.3×
PARSEC 2.1	2.9×	8.4×	4.6×	6.1×

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