

# Profiling, Optimization, and Beyond!

Ludovic Courtès  
Emmanuel Jeanvoine

INRIA SED  
sed-bordeaux@inria.fr

# Outline

## 1 Introduction

## 2 Tools

- Profiling Tools for C, C++, and More
- Java & co.
- Other Tools

## 3 Optimization

## 4 Conclusion



# Why Profile and Optimize?

## Rationale

- Computing power is finite
- Small parts can slow down the entire application

## Process

- 1 Identify bottlenecks (trying to guess what's slow)
- 2 Choose better algorithms or improve the implementation (*optimization*)



# Why Profile and Optimize?

## Rationale

- Computing power is finite
- Small parts can slow down the entire application

## Process

- 1 Identify bottlenecks (*profiling*)
- 2 Choose better algorithms or improve the implementation (*optimization*)



# Why Profile and Optimize?

## Rationale

- Computing power is finite
- Small parts can slow down the entire application

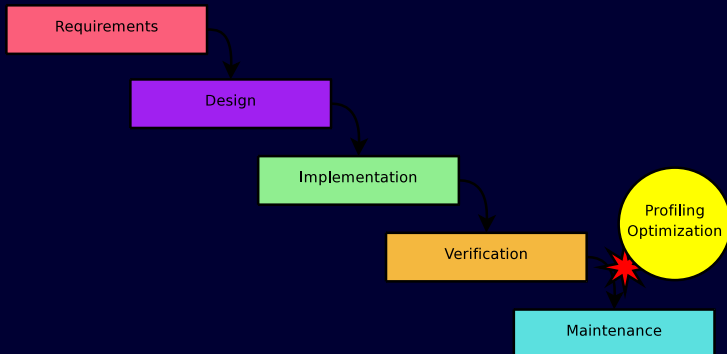
## Process

- 1 Identify bottlenecks (*profiling*)
- 2 Choose better algorithms or improve the implementation (*optimization*)

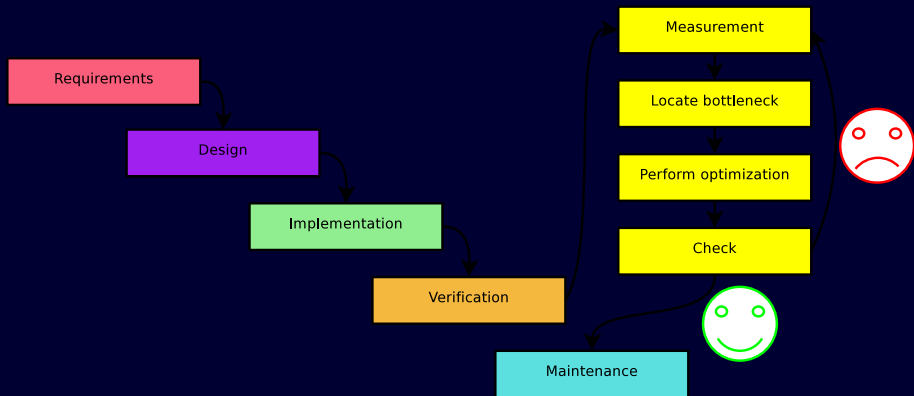


# Optimization in the Development Process

## The “Waterfall” Model



# Optimization in the Development Process



# Optimization in the Development Process

## At a Glance

Optimizations must be performed

- On fully implemented and tested code
- On a version optimized by the compiler (*release*, aka. `-O3`)
- Using representative input data





# How to Profile?

## The Hard Way

- `printf ("%i", time (NULL));`



# How to Profile?

## The Hard Way

- `printf ("%i", time (NULL));`

## How Profilers Do It

- Call stack sampling
- Optional function call instrumentation
- Hardware simulation
- Hardware counters



# Outline

## 1 Introduction

## 2 Tools

- Profiling Tools for C, C++, and More
- Java & co.
- Other Tools

## 3 Optimization

## 4 Conclusion

# Outline

## 1 Introduction

## 2 Tools

- Profiling Tools for C, C++, and More
  - Java & co.
  - Other Tools

## 3 Optimization

## 4 Conclusion

# GNU gprof, A Statistical Profiler

<http://www.gnu.org/software/binutils/>

## How It Works

- Samples the call stack at regular intervals
- Estimates the time spent in each function
- Records the number of calls and call sites of each function



# GNU gprof, A Statistical Profiler

<http://www.gnu.org/software/binutils/>

## How It Works

- Samples the call stack at regular intervals
- Estimates the time spent in each function
- Records the number of calls and call sites of each function

## Using gprof

- Compile with `-pg`: `gcc -pg -o foo foo.c`
- Run it: yields a `gmon.out` file
- View the profile with `gprof`



# GNU gprof: Example

## Flat Profile

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
98.55	0.68	0.68	195615	0.00	0.00	compute_cell
1.45	0.69	0.01	1	10.00	690.00	multiply
0.00	0.69	0.00	1	0.00	0.00	initialize



# GNU gprof: Example

## Flat Profile

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
98.55	0.68	0.68	195615	0.00	0.00	compute_cell
1.45	0.69	0.01	1	10.00	690.00	multiply
0.00	0.69	0.00	1	0.00	0.00	initialize

## Call Graph

index	% time	self	children	called	name
		0.01	0.68	1/1	main [2]
[1]	100.0	0.01	0.68	1	multiply [1]
		0.68	0.00	195615/195615	compute_cell [3]
-----					
					<spontaneous>
[2]	100.0	0.00	0.69		main [2]
		0.01	0.68	1/1	multiply [1]
		0.00	0.00	1/1	initialize [4]





# GNU gprof: Annotated Source

```
        :unsigned long
        :scm_ihashv (SCM obj, unsigned long n)
548 0.0589 :{ /* scm_ihashv total: 15330 1.6480 */
  11 0.0012 : if (SCM_CHARP(obj))
        :   return (scm_c_lowercase (SCM_CHAR (obj))) % n;
        :
235 0.0253 : if (SCM_NUMP(obj))
        :   return (unsigned long) scm_hasher (obj, n, 10);
        : else
257 0.0276 :   return SCM_UNPACK (obj) % n;
14279 1.5350 :}
```

## Disadvantages

- Needs recompilation
- Instrumentation adds function call overhead
- Can't profile code in shared libraries
- Profiles only the main thread



# Overview of Valgrind

<http://valgrind.org/>

## Features

- CPU simulator with several “tools”
- memcheck (memory error checker), massif (heap profiler)
- cachegrind and callgrind (cache profilers)

## Application Profiling with callgrind

- Apps **don't need to be recompiled!**
- Works with **shared libraries!**
- Works with **multi-threaded code!**
- **Detailed info:** execution time, cache misses, etc.
- **Nice GUI:** KCachegrind



# Overview of Valgrind

<http://valgrind.org/>

## Features

- CPU simulator with several “tools”
- memcheck (memory error checker), massif (heap profiler)
- cachegrind and callgrind (cache profilers)

## Application Profiling with callgrind

- Apps **don't need to be recompiled!**
- Works with **shared libraries!**
- Works with **multi-threaded code!** (... but runs on 1 core)
- **Detailed info:** execution time, cache misses, etc.
- **Nice GUI:** KCachegrind
- ... but is quite **slow** ( $\approx 20\text{--}100\times$  slower)



# Running an Application with Valgrind/Callgrind

```
$ valgrind --tool=callgrind  
                                     \  
                                     \  
                                     \  
                                     \  
my-application and its arguments
```



# Running an Application with Valgrind/Callgrind

```
$ valgrind --tool=callgrind \
    --simulate-cache=yes \
    \
    \
    my-application and its arguments
```



# Running an Application with Valgrind/Callgrind

```
$ valgrind --tool=callgrind \
    --simulate-cache=yes \
    --dump-instr=yes \
    my-application and its arguments
```



# Running an Application with Valgrind/Callgrind

```
$ valgrind --tool=callgrind \
    --simulate-cache=yes \
    --dump-instr=yes \
    --separate-threads=yes \
    my-application and its arguments
```





# Exploiting Callgrind's Profiles with KCachegrind

The screenshot shows the KCachegrind application interface. At the top, there is a menu bar (File, View, Go, Settings, Help) and a toolbar with navigation icons. The main window is titled "Instruction Fetch" and displays a call graph for the function "canonicalize\_define".

The call graph is organized into tabs: Types, Callers, All Callers, Source, and Callee Map. The Callee Map tab is active, showing a hierarchical view of function calls. The root node is "canonicalize\_define", which calls "scm\_i\_sweep\_solid", "scm\_gc\_mark", "scm\_i\_gc", "scm\_i\_c\_mem2sys", "scm\_mark\_all", "scm\_from\_locale", "scm\_gc\_mark", "scm\_i\_c\_make\_sys", "ceval'2 <cycle 3>", "scm\_m\_define", "scm\_i\_sweep\_solid", "scm\_list\_1", "scm\_read\_sexp'2", and "canonicalize\_define".

Below the call graph, the "Assembler" tab is active, showing assembly instructions. The instructions are listed in a table with columns for instruction number, instruction register (lr), hex value, and assembler code.

#	lr	Hex	Assembler
3 2082	0.00	e8 01 c8 1e ff	call 2088 <scm_com...
	0.18		330 calls to 'sc...
	0.00		1 call to '_dl_ru...
3 2087	0.00	89 04 24	mov %eax,(%esp)
3 208A	0.00	e8 89 f8 fe ff	call 21918 <scm_list_...

At the bottom of the window, a status bar displays: "callgrind.out.26649 [1] - Total Instruction Fetch Cost: 35 602 310".

# Exploiting Callgrind's Profiles with KCachegrind

The screenshot shows the KCachegrind interface. The top menu bar includes File, View, Go, Settings, and Help. Below the menu is a toolbar with navigation icons. The main window is titled 'canonicalize\_define' and has tabs for Types, Callers, All Callers, Source, and Callee Map. The Source tab is active, displaying assembly code for the function. A call graph is overlaid on the assembly, showing a call to 'scm\_i\_remove\_weaks\_from\_w...' with a cost of 11.21%. The assembly code is as follows:

```
# D1mr Hex Assembler
3 2087 89 04 24 mov %eax,(%esp)
3 208A 0.00 e8 89 f8 fe ff call 21918 <scm_list_10>
3 208F 0.00 8b 36 mov (%esi),%esi
```

The bottom status bar indicates: callgrind.out.26649 [1] - Total L1 Data Read Miss Cost: 133 571

Incl.	Self	Called	Function
93.55	0.00	1	(below main)
93.55	0.00	1	main
93.53	0.00	1	scm_boot_guile
93.52	0.00	1	scm_with_guile
93.52	0.00	1	scm_i_with_guile_e
93.43	8.76	2	<cycle 3>
59.93	1.80	29 300	<cycle 2>
52.97	0.04	3	scm_i_gc <cycle 2
50.67	0.03	3	scm_mark_all
29.61	0.11	8 827	scm_i_c_mem2syn
28.65	0.90	12 125	scm_read_expres
27.07	0.00	7 174	scm_from_locale_
26.73	26.35	9	scm_i_mark_weak
26.58	0.19	463	scm_m_define <c
25.74	0.07	474	canonicalize_defin

# Exploiting Callgrind's Profiles with KCachegrind

File View Go Settings Help

Instruction Fetch

Search: (No Grouping)

Incl.	Self	Cal	Function
99.98	18	8	foo.omp_fn.0
87.48	0.00	7	start_thread
87.48	0.00	7	gomp_thread_start
12.51	0.00	(0)	0x00000000000000a9c
12.50	0.00	1	0x00000000000040077c
12.50	0.00	1	(below main)
12.50	0.00	1	main
12.50	0.00	1	foo
0.01	0.01	(0)	profil_counter
0.01	0.00	1	_dl_start
0.01	0.00	1	_dl_sysdep_start
0.01	0.00	1	dl_main
0.01	0.00	...3	_dl_lookup_symbol_x
0.01	0.00	...3	do_lookup_x
0.01	0.00	8	_dl_relocate_object
0.00	0.00	1	_dl_init
0.00	0.00	9	call_init
0.00	0.00	...2	check_match.12089
0.00	0.00	43	_dl_runtime_resolve
0.00	0.00	43	_dl_fixup
0.00	0.00	1	0x0000000000055ac2c8

foo.omp\_fn.0

Types	Callers	All Callers	Source	Callee Map
Event Type	Incl.	Self	Short	Formula
Instruction Fetch	99.98	99.98	lr	
Data Read Access	99.97	99.97	Dr	
Data Write Access	99.99	99.99	Dw	
L1 Instr. Fetch Miss	8.68	0.16	l1mr	

Caller Map

Parts

Call Graph

Callees

All Callees

Assembler

callgrind.out.19668 [1] - Total Instruction Fetch Cost: 3 200 769 687

# Exploiting Callgrind's Profiles with KCachegrind

File View Go Settings Help

Instruction Fetch

### foo.omp\_fn.0

Types Callers All Callers Source Callee Map

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	99.88	99.88	lr	

Search: (No Grouping)

Incl.	Self	Called	Function
99.88	99.88	1	foo.omp_fn.0
0.07	0.00	1	_dl_start
0.07	0.00	1	_dl_sysdep_start
0.07	0.00	1	dl_main
0.07	0.01	150	_dl_lookup_symbol
0.06	0.03	150	do_lookup_x
0.05	0.01	8	_dl_relocate_object
0.03	0.00	1	_dl_init
0.03	0.00	9	call_init
0.02	0.02	7051	check_match

main (99.88%)  
foo (99.88%)  
foo.omp\_fn.0 (99.88%)

Caller Map Parts Call Graph Callees All Callees Assembler

callgrind.out.9450 [1] - Total Instruction Fetch Cost: 400 497 274

# Exploiting Callgrind's Profiles with KCachegrind

File View Go Settings Help

Instruction Fetch

### foo.omp\_fn.0

Types Callers All Callers Source Callee Map

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	100.00	100.00	lr	

Search: (No Grouping)

Incl.	Self	Called	Function
100.00	100.00	1	foo.omp_fn.0
0.01	0.01	(0)	profil_counter
0.00	0.00	2	gomp_barrier_
0.00	0.00	2	gomp_barrier_
0.00	0.00	1	printf
0.00	0.00	1	vfprintf
0.00	0.00	1	_l_lock_696
0.00	0.00	1	gomp_team_t
0.00	0.00	1	gomp_team_t
0.00	0.00	3	IO file xsubtr

start\_thread (100.00%)  
↓ (100.00%)  
gomp\_thread\_start (100.00%)  
↓ (100.00%)  
foo.omp\_fn.0 (100.00%)

Caller Map Parts Call Graph Callees All Callees Assembler

callgrind.out.9450 [2] - Total Instruction Fetch Cost: 400 037 583

# PAPI, a Portable, Embeddable Profiler

<http://icl.cs.utk.edu/papi/>

## Features

- Library + tools to collect *hardware counters*
- Examples: instructions executed, cache misses, etc.
- Portable: Linux, AIX, Solaris, IRIX, Unicos, Catamount
- On Linux, uses the (non-standard) `perfmon2` kernel module



# PAPI, a Portable, Embeddable “Profiler”

<http://icl.cs.utk.edu/papi/>

## Features

- Library + tools to collect *hardware counters*
- Examples: instructions executed, cache misses, etc.
- Portable: Linux, AIX, Solaris, IRIX, Unicos, Catamount
- On Linux, uses the (non-standard) perfmon2 kernel module

Low-level profiling ahead!



# PAPI: Listing Available Hardware Counters

```
$ papi_avail
```



# PAPI: Listing Available Hardware Counters

```
$ papi_avail
```

Name	Code	Avail	Deriv	Description (Note)
PAPI_L1_DCM	0x80000000	Yes	No	Level 1 data cache misses
PAPI_L1_ICM	0x80000001	Yes	No	Level 1 instruction cache misses

# PAPI: Listing Available Hardware Counters

```
$ papi_avail
```

Name	Code	Avail	Deriv	Description (Note)
PAPI_L1_DCM	0x80000000	Yes	No	Level 1 data cache misses
PAPI_L1_ICM	0x80000001	Yes	No	Level 1 instruction cache misses
PAPI_TOT_INS	0x80000032	Yes	No	Instructions completed
PAPI_TOT_CYC	0x8000003b	Yes	No	Total cycles

# PAPI: Listing Available Hardware Counters

```
$ papi_avail
```

Name	Code	Avail	Deriv	Description (Note)
PAPI_L1_DCM	0x80000000	Yes	No	Level 1 data cache misses
PAPI_L1_ICM	0x80000001	Yes	No	Level 1 instruction cache misses
PAPI_TOT_INS	0x80000032	Yes	No	Instructions completed
PAPI_TOT_CYC	0x8000003b	Yes	No	Total cycles
PAPI_BR_INS	0x80000037	Yes	No	Branch instructions
PAPI_BR_MSP	0x8000002e	Yes	No	Conditional branch instructions

# PAPI: Listing Available Hardware Counters

```
$ papi_avail
```

Name	Code	Avail	Deriv	Description (Note)
PAPI_L1_DCM	0x80000000	Yes	No	Level 1 data cache misses
PAPI_L1_ICM	0x80000001	Yes	No	Level 1 instruction cache misses
PAPI_TOT_INS	0x80000032	Yes	No	Instructions completed
PAPI_TOT_CYC	0x8000003b	Yes	No	Total cycles
PAPI_BR_INS	0x80000037	Yes	No	Branch instructions
PAPI_BR_MSP	0x8000002e	Yes	No	Conditional branch instructions
PAPI_FP_OPS	0x80000066	Yes	No	Floating point operations

```
...
```

# PAPI: Whole-Program Profiling

```
$ papiex -e PAPI_TOT_INS -e PAPI_TOT_CYC some-program  
...
```



# PAPI: Whole-Program Profiling

```
$ papiex -e PAPI_TOT_INS -e PAPI_TOT_CYC some-program  
...  
$ cat some-program.papiex.host.1234
```



# PAPI: Whole-Program Profiling

```
$ papiex -e PAPI_TOT_INS -e PAPI_TOT_CYC some-program
```

```
...
```

```
$ cat some-program.papiex.host.1234
```

```
...
```

```
PAPI_TOT_INS ..... 271574  
PAPI_TOT_CYC ..... 754512
```



# PAPI: Whole-Program Profiling

```
$ papiex -e PAPI_TOT_INS -e PAPI_TOT_CYC some-program
```

```
...
```

```
$ cat some-program.papiex.host.1234
```

```
...
```

```
PAPI_TOT_INS ..... 271574  
PAPI_TOT_CYC ..... 754512
```

## Pros and Cons

- Easy to use, no program modification, no recompilation
- Coarse-grain





# PAPI: Embedded in a Program

```
for (i = 0; i < N; i++)  
  for (j = 0; j < M; j++) {  
    double sum = 0;  
  
    for (l = 0; l < P; l++)  
      sum += a[i][l] * b[l][j];  
    c[i][j] = sum;  
  
  }
```



# PAPI: Embedded in a Program

```
static int events[] = { PAPI_TOT_INS, PAPI_L1_DCM };

for (i = 0; i < N; i++)
  for (j = 0; j < M; j++) {
    double sum = 0;
    long long counters[2];
    PAPI_start_counters (events, 2);
    for (l = 0; l < P; l++)
      sum += a[i][l] * b[l][j];
    c[i][j] = sum;
    PAPI_stop_counters (counters, 2);
    printf ("L1 data cache misses: %lli\n", counters[0]);
  }
```



# Outline

## 1 Introduction

## 2 Tools

- Profiling Tools for C, C++, and More
- Java & co.
- Other Tools

## 3 Optimization

## 4 Conclusion



# JVM Profiling Tools

Java, Scala, Groovy, etc.

- **HProf**, built-in profiling tool (JVM-TI demo):

```
java -agentlib:hprof=cpu=sample com.example.Class
```



# JVM Profiling Tools

Java, Scala, Groovy, etc.

- **HProf**, built-in profiling tool (JVM-TI demo):  
`java -agentlib:hprof=cpu=sample com.example.Class`
- **JRat** (Java Runtime Analysis Toolkit),  
<http://jrat.sourceforge.net/>

The screenshot shows the JRat Desktop application window. The main area displays a table of profiling data for the class 'Matrice'. The table has columns for Class, Method, Exits, Total ms, Average ms, Total Method ms, Method Time %, and Average Method ms. The data is as follows:

Class	Method	Exits	Total ms	Average ms	Total Method ms	Method Time %	Average Method ms
Matrice	main(String[])	1	557 980	557 980.00	1	.0	1.00
Matrice	Saisie(int,int)	2	51 250	25 625.00	26 766	4.8	13 383.00
Matrice	DonneDouble()	18 000 000	24 484	0.00	24 484	4.4	0.00
Matrice	Somme(Matr...	1	416	416.00	416	.1	416.00
Matrice	Produit(Mat...	1	506 313	506 313.00	506 313	98.7	506 313.00

On the left side, there are several panels: 'Tasks' with sorting options, 'Somme(Matrice)' showing summary statistics (Total Exits: 1, Method Time: 416ms (0.1%), Exceptions: 0, Exception Rate: 0.0%), and 'Session Details' (Start: 18/09/08 10:41:12 CEST, End: 18/09/08 10:50:31 CEST, Duration: 558195 ms, Host: tostaky, Address: 127.0.1.1).

7M of 246M

# JVM Profiling Tools

Java, Scala, Groovy, etc.

- **HProf**, built-in profiling tool (JVM-TI demo):  
`java -agentlib:hprof=cpu=sample com.example.Class`
- **JRat** (Java Runtime Analysis Toolkit),  
<http://jrat.sourceforge.net/>
- **TPTP** (Test and Performance Tools Platform)  
<http://www.eclipse.org/tptp/>

The screenshot shows the Eclipse IDE interface with the Profiling Monitor and Memory Statistics views. The Profiling Monitor view shows the current profile for 'Matrice at tostaky [PID: 12221]'. The Memory Statistics view displays a table of memory usage for the highest 10 total size objects.

Class Name	Package	Live Instanc	Active Size	Total Instar	<Total Size	Avg. Age
double[][]	(default packag	3	12048	4	16064	1.75
char[]	(default packag	6	744	79	7936	1.3
byte[]	(default packag	0	0	8	536	1
Matrice	(default packag	3	72	4	96	2.5

# Outline

## 1 Introduction

## 2 Tools

- Profiling Tools for C, C++, and More
- Java & co.
- Other Tools

## 3 Optimization

## 4 Conclusion

# Profilers for Other Languages

Python	Hotshot <a href="http://docs.python.org/lib/profile.html">http://docs.python.org/lib/profile.html</a>
Ruby	Ruby-Prof <a href="http://ruby-prof.rubyforge.org">http://ruby-prof.rubyforge.org</a>
Perl	Dprof <a href="http://search.cpan.org/~ilyaz/DProf/">http://search.cpan.org/~ilyaz/DProf/</a>
.NET	Mono's profiler <a href="http://mono-project.com/Performance_Tips">http://mono-project.com/Performance_Tips</a>
OCaml	OCamlProf <a href="http://caml.inria.fr/">http://caml.inria.fr/</a>
Scheme	bglprof (Bigloo), statprof (GNU Guile), ...
PHP	Xdebug <a href="http://www.xdebug.org">http://www.xdebug.org</a>



# System-Wide Profiling

## Why system-wide?

- Functionality is spread over **several processes**



# System-Wide Profiling

## Why system-wide?

- Functionality is spread over **several processes**
  - ▶ Client & server (RPC, X11, etc.)
  - ▶ MPI program (several processes)



# System-Wide Profiling

## Why system-wide?

- Functionality is spread over **several processes**
  - ▶ Client & server (RPC, X11, etc.)
  - ▶ MPI program (several processes)
  - ▶ Application dependent on in-kernel functionality



# System-Wide Profiling

## Why system-wide?

- Functionality is spread over **several processes**
  - ▶ Client & server (RPC, X11, etc.)
  - ▶ MPI program (several processes)
  - ▶ Application dependent on in-kernel functionality
- Other profiling methods fail :-)



# System-Wide Profiling

## Why system-wide?

- Functionality is spread over **several processes**
  - Client & server (RPC, X11, etc.)
  - MPI program (several processes)
  - Application dependent on in-kernel functionality
- Other profiling methods fail :-)

## What info does it provide?

- Time spent in processes, functions, kernel code
- Call graphs down into the kernel
- Hardware event counts *à la* PAPI



# System-Wide Profiling

## Why system-wide?

- Functionality is spread over **several processes**
  - Client & server (RPC, X11, etc.)
  - MPI program (several processes)
  - Application dependent on in-kernel functionality
- Other profiling methods fail :-)

## What info does it provide?

- Time spent in processes, functions, kernel code
- Call graphs down into the kernel
- Hardware event counts *à la* PAPI

Applications don't need to be recompiled!



# System-Wide Profiling

## Why system-wide?

- Functionality is spread over **several processes**
  - Client & server (RPC, X11, etc.)
  - MPI program (several processes)
  - Application dependent on in-kernel functionality
- Other profiling methods fail :-)

## What info does it provide?

- Time spent in processes, functions, kernel code
- Call graphs down into the kernel
- Hardware event counts *à la* PAPI

Applications don't need to be recompiled!

Tools: OProfile, SysProf, SystemTap (Linux)



# System-Wide Profiling with OProfile

<http://oprofile.sourceforge.net/>

```
$ sudo opcontrol --init
```





# System-Wide Profiling with OProfile

<http://oprofile.sourceforge.net/>

```
$ sudo opcontrol --init  
$ sudo opcontrol --start
```



# System-Wide Profiling with OProfile

<http://oprofile.sourceforge.net/>

```
$ sudo oprofile --init  
$ sudo oprofile --start  
  
$ run my favorite application(s)...  
$ sudo oprofile --stop
```



# System-Wide Profiling with OProfile

<http://oprofile.sourceforge.net/>

```
$ sudo opcontrol --init  
$ sudo opcontrol --start  
  
$ run my favorite application(s)...  
$ sudo opcontrol --stop  
  
$ oprofile -gdf | op2cg  
$ kcachegrind oprof.out.*
```



# System-Wide Profiling with OProfile

<http://oprofile.sourceforge.net/>

```
$ sudo opcontrol --init
```

```
$ sudo opcontrol --start
```

```
$ run my favorite application(s)...
```

```
$ sudo opcontrol --stop
```

```
$ opreport --merge all
```

```
samples|      %|
```

```
-----
```

```
961251 73.7464 libguile-2.0.so.18.0.0
```

```
95751  7.3459 vmlinux
```

```
61499  4.7182 ld-2.11.1.so
```

```
35647  2.7348 emacs
```

```
32005  2.4554 libgc.so.1.0.3
```

```
...
```



# System-Wide Profiling with OProfile

<http://oprofile.sourceforge.net/>

```
$ sudo opcontrol --init
```

```
$ sudo opcontrol --start
```

```
$ run my favorite application(s)...
```

```
$ sudo opcontrol --stop
```

```
$ opreport --symbols
```

samples	%	image name	symbol name
787730	62.0882	libguile-2.0.so.18.0.0	vm_debug_engine
61207	4.8243	ld-2.11.1.so	__tls_get_addr
23477	1.8504	vmlinux	flat_send_IPI_all
21089	1.6622	libguile-2.0.so.18.0.0	scm_hash_fn_create_h
15330	1.2083	libguile-2.0.so.18.0.0	scm_ihashv
14524	1.1448	libguile-2.0.so.18.0.0	scm_ihashq

```
...
```

# System-Wide Profiling with OProfile

<http://oprofile.sourceforge.net/>

```
$ sudo opcontrol --init
```

```
$ sudo opcontrol --start
```

```
$ run my favorite application(s)...
```

```
$ sudo opcontrol --stop
```

```
$ opreport -g --symbols libguile-2.0.so.18.0.0
```

samples	%	linenr	info	symbol name
787730	84.6828	vm-engine.c:38		vm_debug_engine
21089	2.2671	hashtab.c:518		scm_hash_fn_create_h
15330	1.6480	hash.c:214		scm_ihashv
14524	1.5614	hash.c:184		scm_ihashq
13060	1.4040	vm.c:598		scm_the_vm
7955	0.8552	vm.c:391		vm_make_boot_program
...				

# System-Wide Profiling with OProfile

<http://oprofile.sourceforge.net/>

```
$ sudo opcontrol --init  
$ sudo opcontrol --start -e MISALIGN_MEM_REF : 700 -e ...  
  
$ run my favorite application(s)...  
$ sudo opcontrol --stop
```

hardware counter

sampling period

# Proprietary Profilers

- Intel VTune™ (MS Visual Studio, GNU/Linux)
- AMD CodeAnalyst™ (Windows, GNU/Linux)



# Proprietary Profilers

- Intel VTune™ (MS Visual Studio, GNU/Linux)
- AMD CodeAnalyst™ (Windows, GNU/Linux)

<https://plafrim.bordeaux.inria.fr/>

→ [francois.rue@inria.fr](mailto:francois.rue@inria.fr), [plafrim-support@inria.fr](mailto:plafrim-support@inria.fr)



# Outline

## 1 Introduction

## 2 Tools

- Profiling Tools for C, C++, and More
- Java & co.
- Other Tools

## 3 Optimization

## 4 Conclusion



# Some Advice

“Free advice: you get what you pay for.”

- “[P]rogrammers are notoriously bad at predicting how their programs actually perform.” (*GCC Manual*)
- “Premature optimization is the root of all evil” (D. E. Knuth)



# Some Advice

“Free advice: you get what you pay for.”

- “[P]rogrammers are notoriously bad at predicting how their programs actually perform.” (*GCC Manual*)
  - ▶ Don’t guess—use a profiler
  - ▶ Focus optimization efforts on the “critical path”
- “Premature optimization is the root of all evil” (D. E. Knuth)



# Some Advice

“Free advice: you get what you pay for.”

- “[P]rogrammers are notoriously bad at predicting how their programs actually perform.” (*GCC Manual*)
  - ▶ Don’t guess—use a profiler
  - ▶ Focus optimization efforts on the “critical path”
- “Premature optimization is the root of all evil” (D. E. Knuth)
  - ▶ Optimize once things “work”
  - ▶ Don’t trade maintainability for micro-optimizations



# Reducing the Algorithm Complexity

## Examples

- Sorting algorithm (quick sort, shell sort, heap sort...)
- String management (concatenation, regexp, ...)
- Numerical methods (ordinary differential equation, integration, stochastic processes, ...)

## Warning:

- Visible effect on large data set
- Choice of algorithm is a space/time tradeoff



# Understanding Memory Locality

Storage Area	Register	L1 Cache	L2 Cache	RAM	Swap
Cycles to Access	$\leq 1$	$\approx 3$	$\approx 14$	$\approx 240$	$\approx 10^7$
Town	Talence	Pessac	Cestas	Toulouse	Mars

# Understanding Memory Locality

Storage Area	Register	L1 Cache	L2 Cache	RAM	Swap
Cycles to Access	$\leq 1$	$\approx 3$	$\approx 14$	$\approx 240$	$\approx 10^7$
Town	Talence	Pessac	Cestas	Toulouse	Mars

## Dealing With It...

- Avoid “remote” accesses on the critical path
- Cachegrind, PAPI, etc. can help
- U. Drepper, *What Every Programmer Should Know About Memory*, <http://people.redhat.com/drepper/>





# The Incredible Effect of Repeated L1 Cache Misses

## Version 1

```
int t[4096][4096];
long a;
for (i = 0; i < 4096; i++)
    for (j = 0; j < 4096; j++)
        a += t[i][j];
```

## Version 2

```
int t[4096][4096];
long a;
for (i = 0; i < 4096; i++)
    for (j = 0; j < 4096; j++)
        a += t[j][i];
```



# The Incredible Effect of Repeated L1 Cache Misses

## Version 1

```
int t[4096][4096];
long a;
for (i = 0; i < 4096; i++)
    for (j = 0; j < 4096; j++)
        a += t[i][j];
```

100 executions: 6.2 s

## Version 2

```
int t[4096][4096];
long a;
for (i = 0; i < 4096; i++)
    for (j = 0; j < 4096; j++)
        a += t[j][i];
```

100 executions: 40.3 s



# The Incredible Effect of Repeated L1 Cache Misses

## Version 1

```
int t[4096][4096];
long a;
for (i = 0; i < 4096; i++)
    for (j = 0; j < 4096; j++)
        a += t[i][j];
```

100 executions: 6.2 s

## Version 2

```
int t[4096][4096];
long a;
for (i = 0; i < 4096; i++)
    for (j = 0; j < 4096; j++)
        a += t[j][i];
```

100 executions: 40.3 s

## What's Wrong?

- L1 cache is 32 KiB, with 64 B cache lines
- when data not cached  $\implies$  CPU pre-fetches 64 B
- Version 2 fetches 64 B for each access but uses only 4 B
- $\implies$  cache thrashing



# Use Optimized Libraries

- General-purpose libraries (C++'s STL, Glib, liboil)
- Arbitrary precision computations (GNU MP, GNU MPFR, MPC)
- Linear algebra (BLAS, LAPACK/SCALAPACK, GNU Scientific Lib.)
- Discrete Fourier transforms (GNU Scientific Library, FFTW)
- Ordinary differential equations (GNU Scientific Library, BiM, CVODE)
- ...



# Code Parallelization

## Easy C/C++/Fortran Parallelization with OpenMP

### 1 Annotate code

```
for(i = 0; i < N; i++)
  for (j = 0; j < M; j++) {
    double sum = 0;
    for (l = 0; l < P; l++)
      sum += a[i][l] * b[l][j];
    c[i][j] = sum;
  }
```

# Code Parallelization

## Easy C/C++/Fortran Parallelization with OpenMP

- 1 Annotate code

```
#pragma omp parallel for private(j,l)
for(i = 0; i < N; i++)
  for (j = 0; j < M; j++) {
    double sum = 0;
    for (l = 0; l < P; l++)
      sum += a[i][l] * b[l][j];
    c[i][j] = sum;
  }
```

# Code Parallelization

## Easy C/C++/Fortran Parallelization with OpenMP

- 1 Annotate code

```
#pragma omp parallel for private(j,l)
for(i = 0; i < N; i++)
  for (j = 0; j < M; j++) {
    double sum = 0;
    for (l = 0; l < P; l++)
      sum += a[i][l] * b[l][j];
    c[i][j] = sum;
  }
```

- 2 Compile: `gcc -c -fopenmp foo.c (GCC 4.2+)`



# Outline

## 1 Introduction

## 2 Tools

- Profiling Tools for C, C++, and More
- Java & co.
- Other Tools

## 3 Optimization

## 4 Conclusion





# Summary

- Optimization implies profiling
- Optimization should be disciplined



# Summary

- Optimization implies profiling
  - ▶ Many good tools exist!
- Optimization should be disciplined



# Summary

- Optimization implies profiling
  - ▶ Many good tools exist!
  - ▶ Many are non-intrusive
- Optimization should be disciplined

# Summary

- Optimization implies profiling
  - ▶ Many good tools exist!
  - ▶ Many are non-intrusive
  - ▶ From coarse-grain (`gprof`) to low-level (Cachegrind, PAPI)
- Optimization should be disciplined

# Summary

- Optimization implies profiling
  - ▶ Many good tools exist!
  - ▶ Many are non-intrusive
  - ▶ From coarse-grain (`gprof`) to low-level (Cachegrind, PAPI)
- Optimization should be disciplined
  - ▶ Avoid “premature optimization”

# Summary

- Optimization implies profiling
  - ▶ Many good tools exist!
  - ▶ Many are non-intrusive
  - ▶ From coarse-grain (`gprof`) to low-level (Cachegrind, PAPI)
- Optimization should be disciplined
  - ▶ Avoid “premature optimization”
  - ▶ Use feedback from the profiling tools

# Summary

- Optimization implies profiling
  - ▶ Many good tools exist!
  - ▶ Many are non-intrusive
  - ▶ From coarse-grain (`gprof`) to low-level (Cachegrind, PAPI)
- Optimization should be disciplined
  - ▶ Avoid “premature optimization”
  - ▶ **Always** use feedback from the profiling tools

# Summary

- Optimization implies profiling
  - ▶ Many good tools exist!
  - ▶ Many are non-intrusive
  - ▶ From coarse-grain (`gprof`) to low-level (Cachegrind, PAPI)
- Optimization should be disciplined
  - ▶ Avoid “premature optimization”
  - ▶ Always use feedback from the profiling tools
  - ▶ Think algorithm before micro-optimization



# That's All Folks!

Questions?

sed-bordeaux@inria.fr

<http://sed.bordeaux.inria.fr/>  
ludovic.courtes@inria.fr

