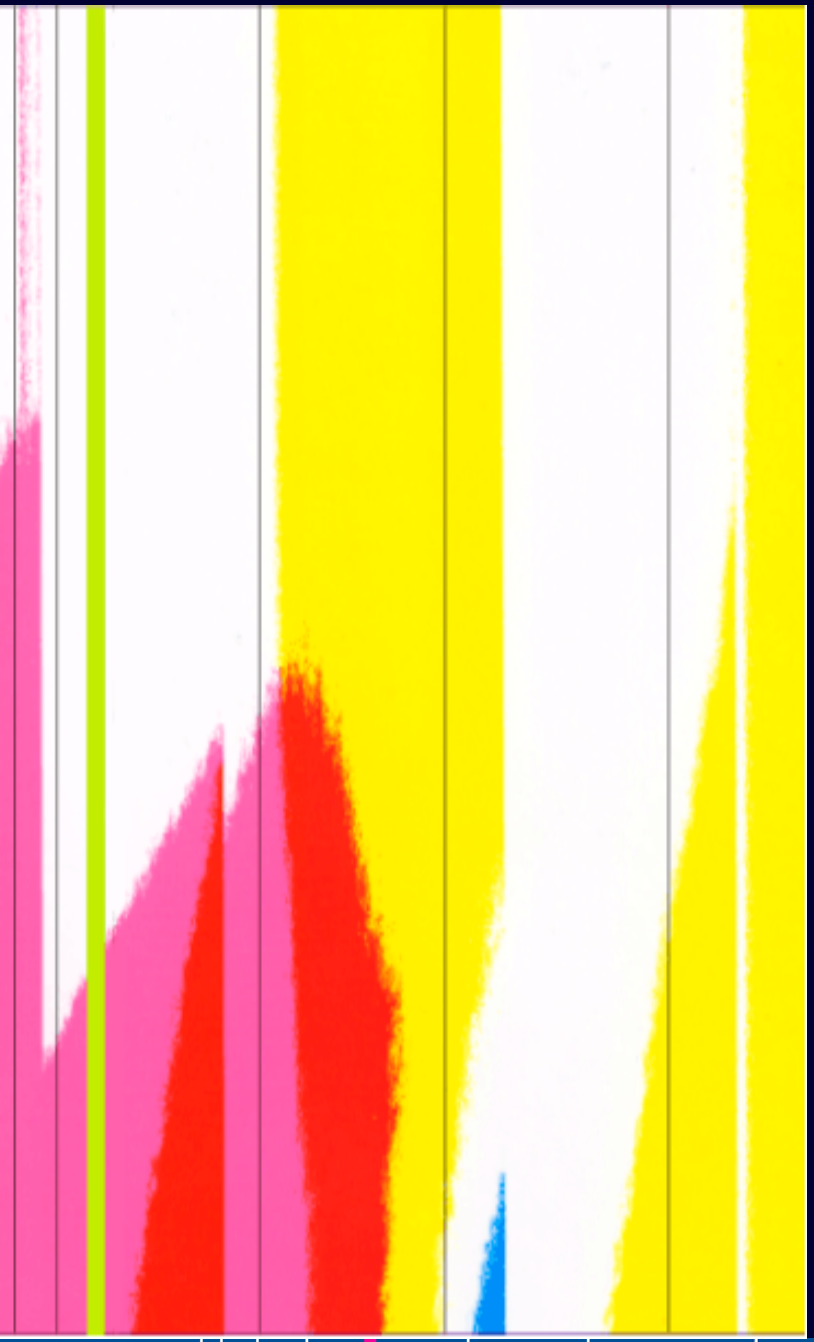


Profiling, Optimization, and Beyond!

Ludovic Courtès
Emmanuel Jeanvoine

INRIA SED
sed-bordeaux@inria.fr

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

Outline

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

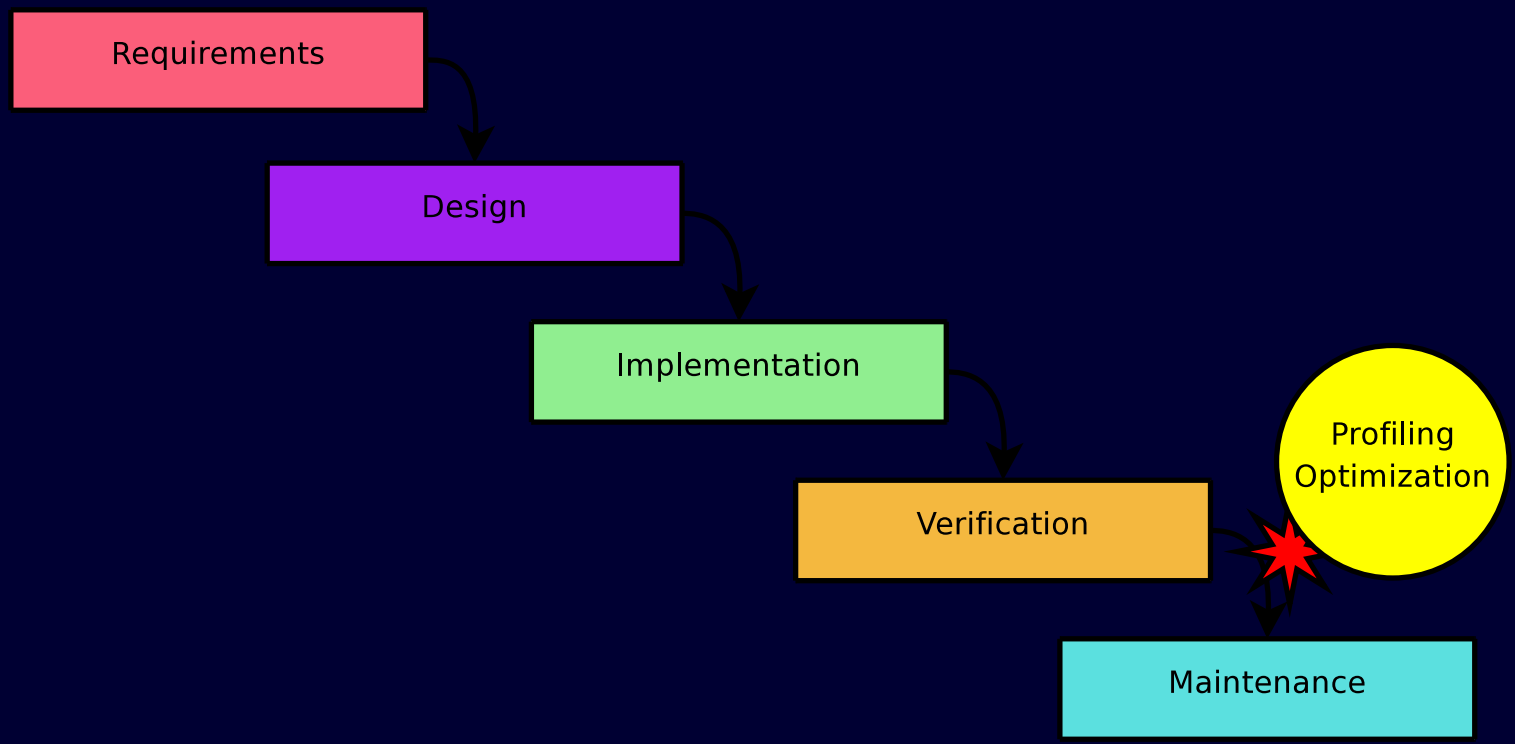
- Computing power is finite
- Small parts can slow down the entire application

Process

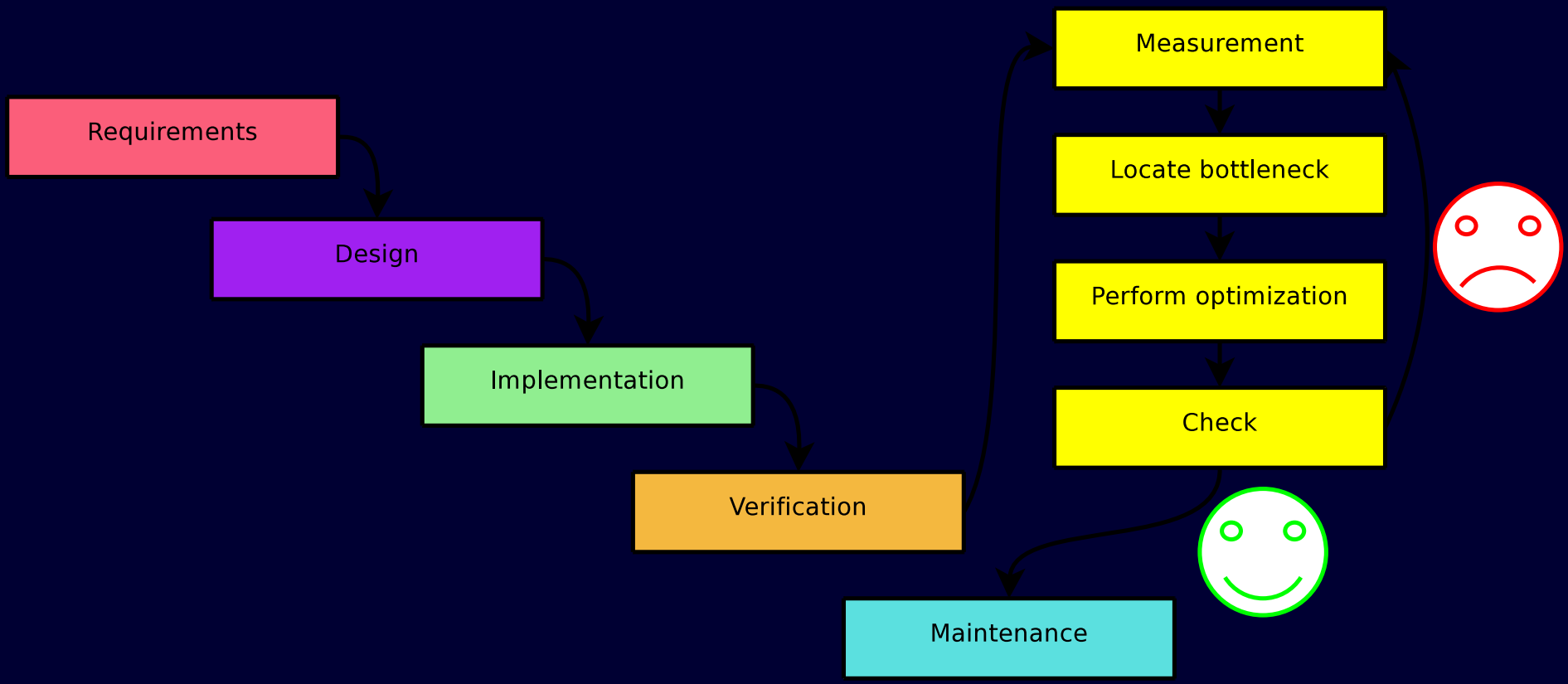
1. Identify bottlenecks (*profiling*)
2. Choose better algorithms or improve the implementation (*optimization*)

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

Outline

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.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  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.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  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_downcase (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 :}
```

GNU gprof: Disadvantages

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** ($\approx$ 20–100$\times$ 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
```

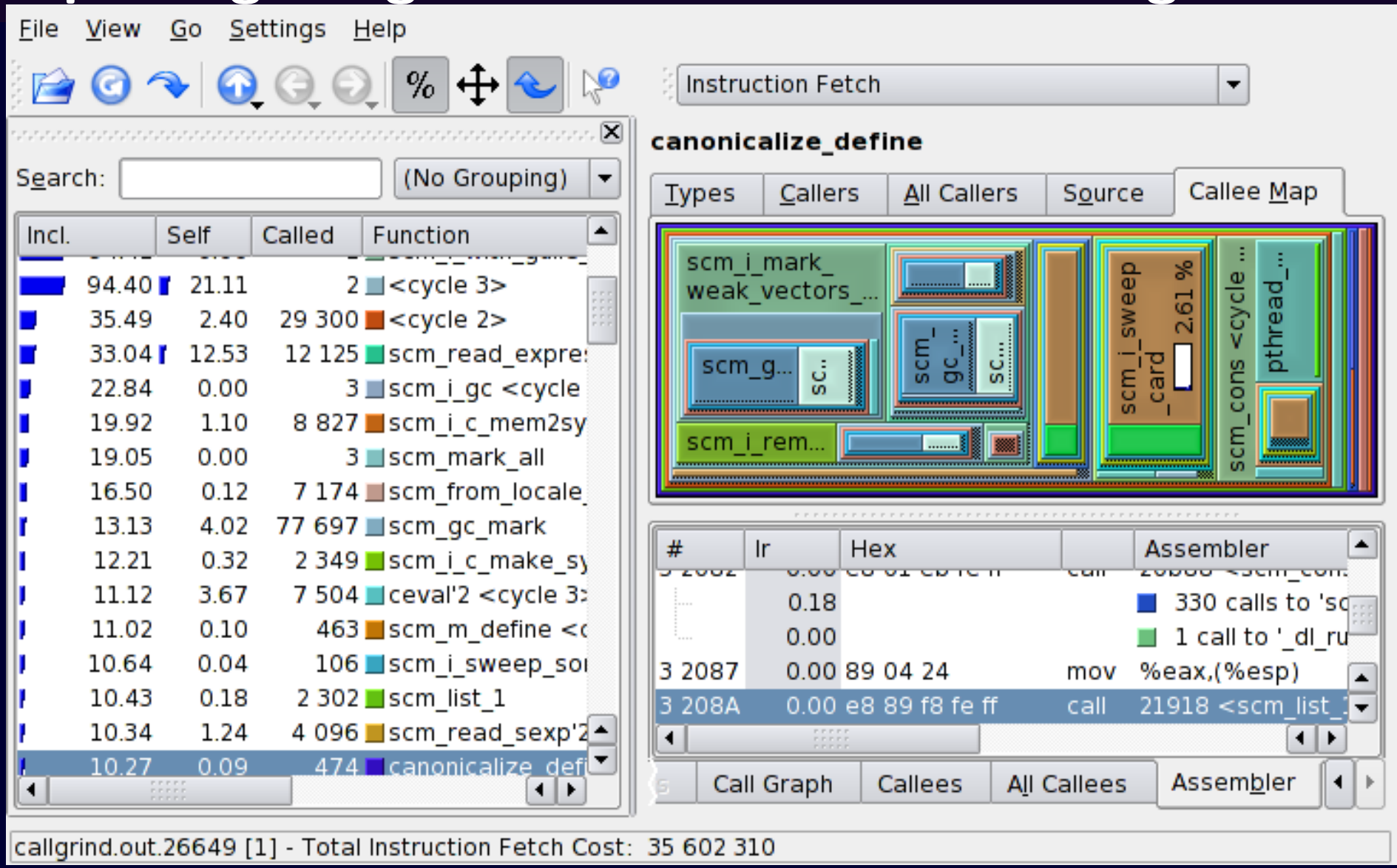
Running an Application with Valgrind/Callgrind

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

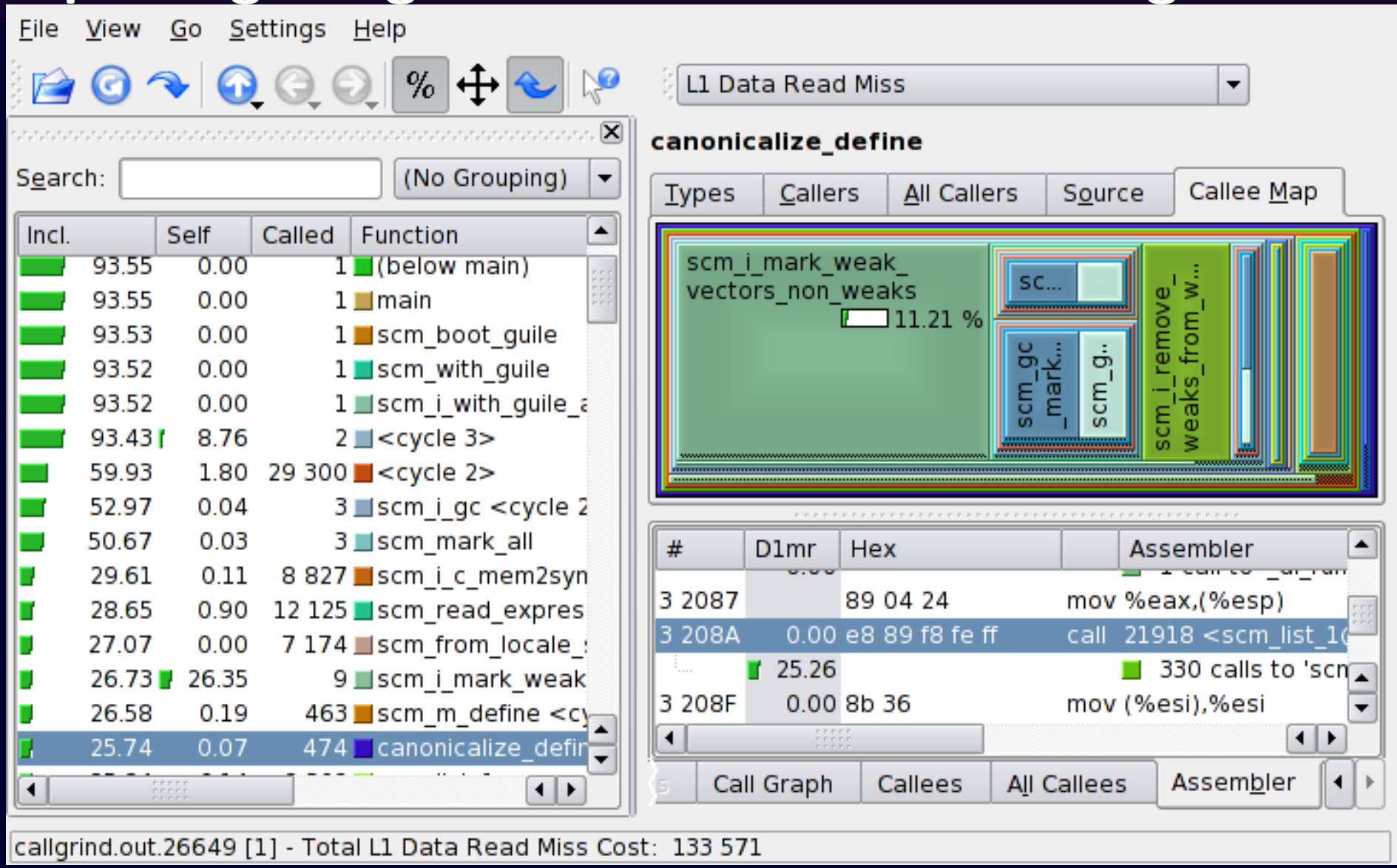
Running an Application with Valgrind/Callgrind

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

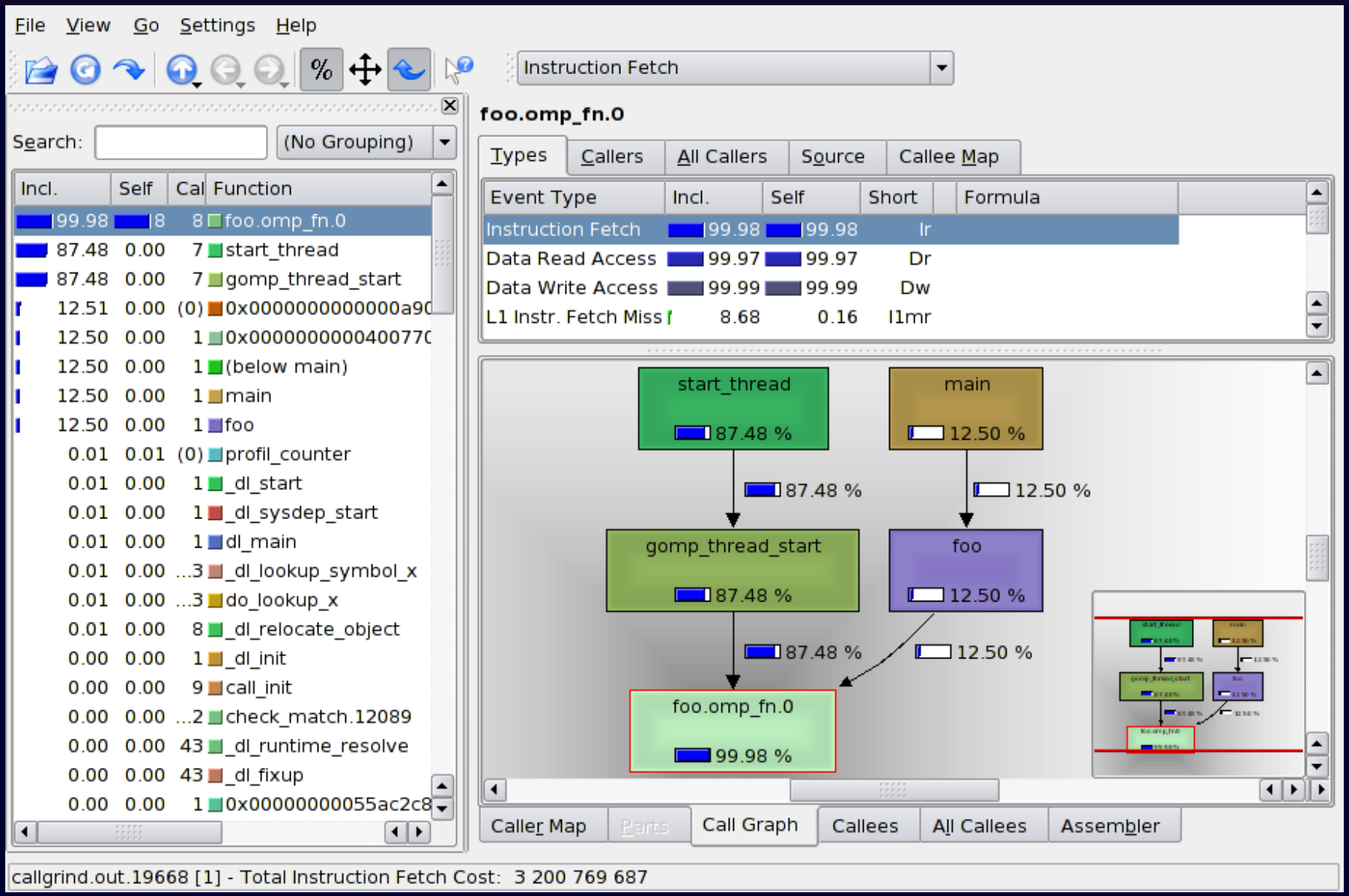
Exploiting Callgrind's Profiles with KCachegrind



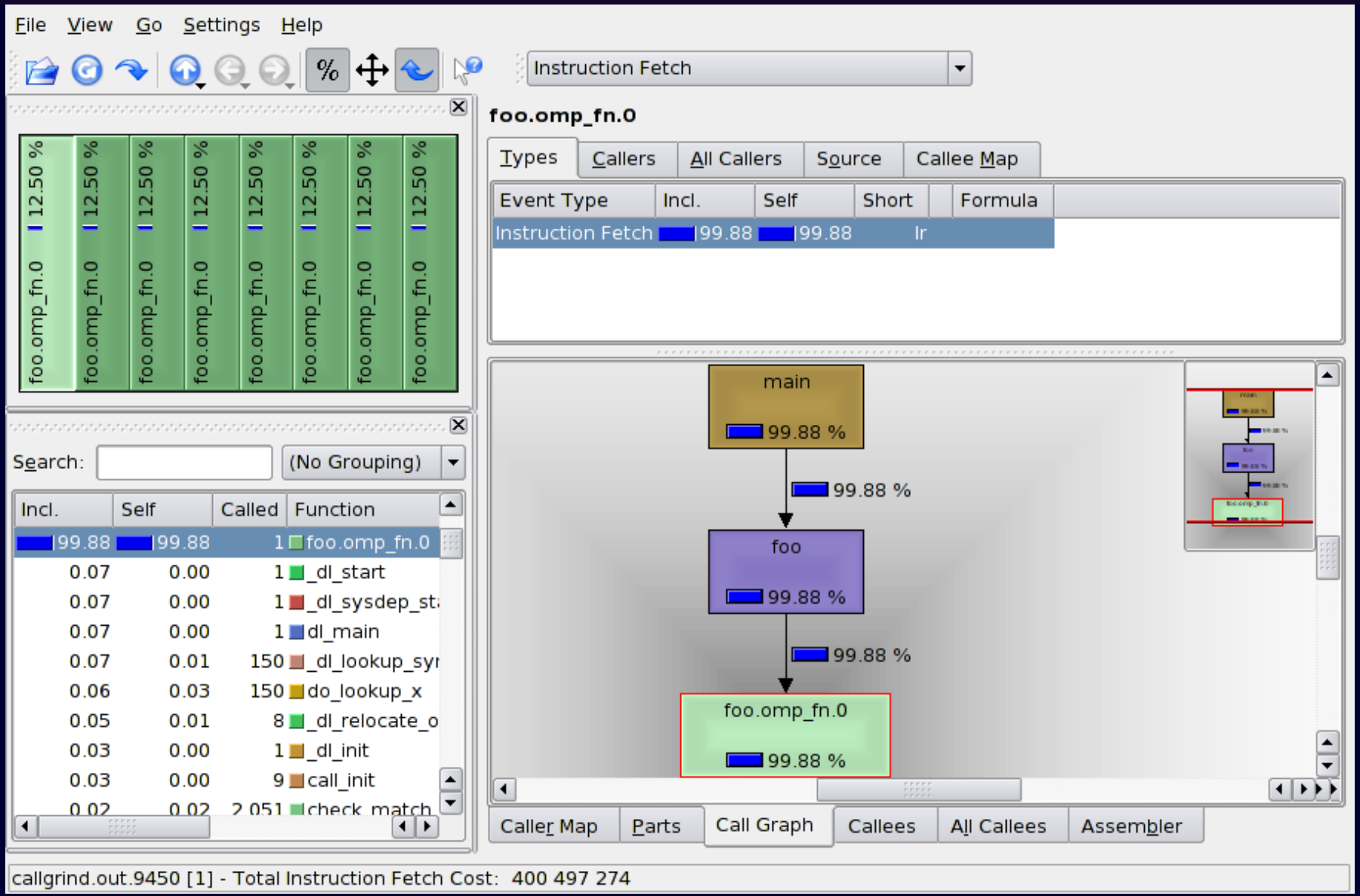
Exploiting Callgrind's Profiles with KCachegrind



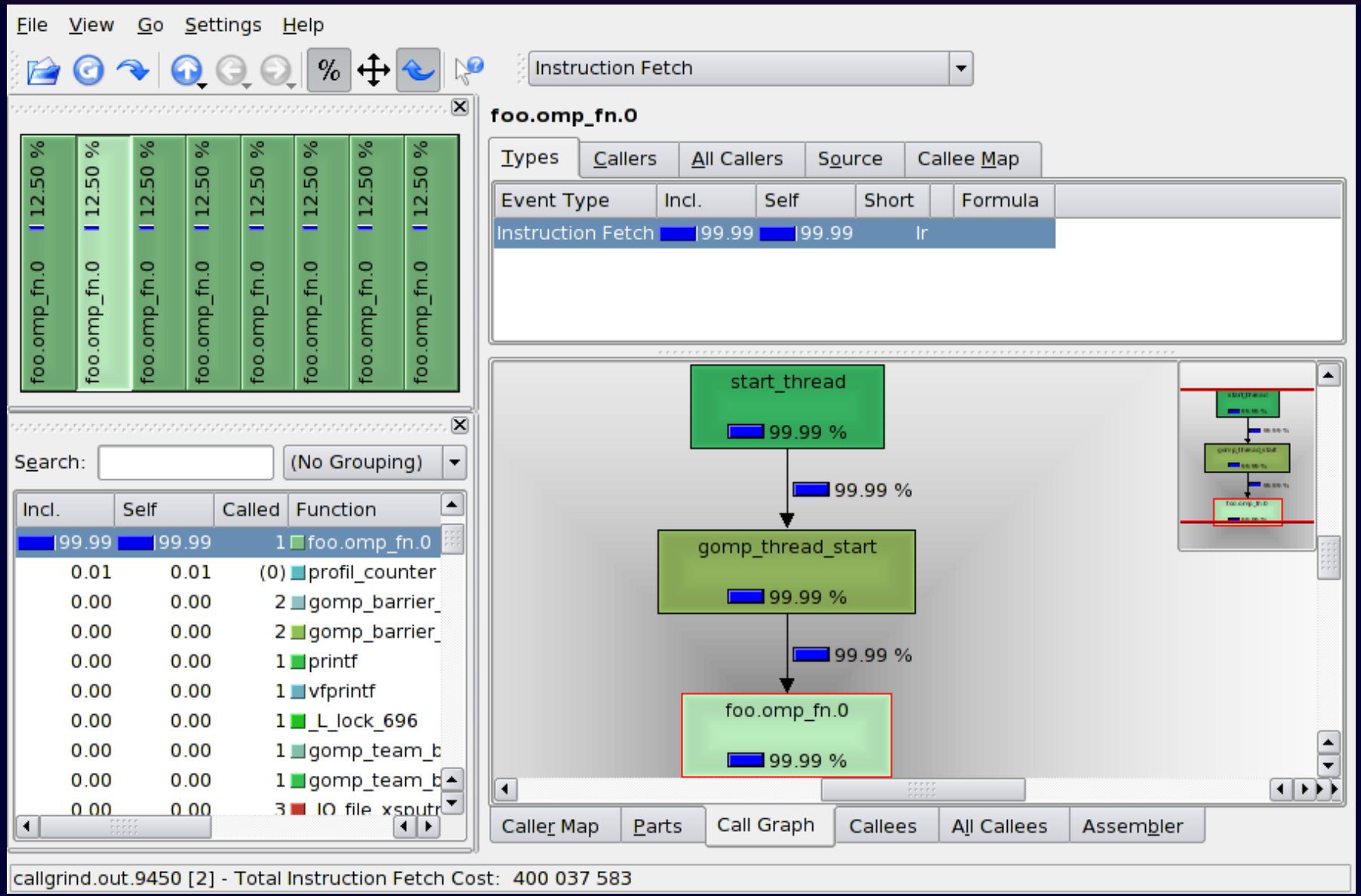
Exploiting Callgrind's Profiles with KCachegrind



Exploiting Callgrind's Profiles with KCachegrind



Exploiting Callgrind's Profiles with KCachegrind



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

JVM Profiling Tools

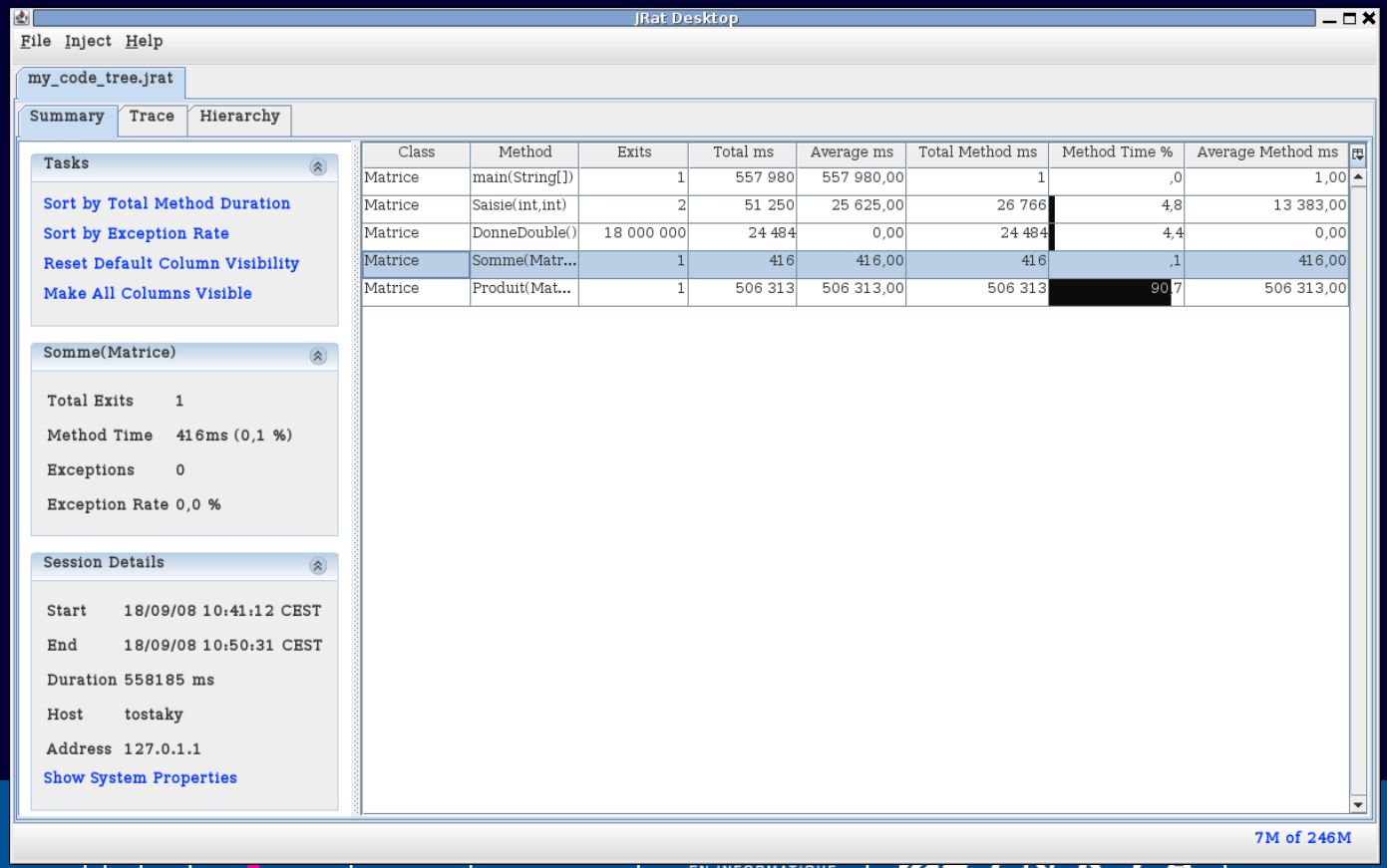
Java, Scala, Groovy, etc.

- **HProf**, built-in profiling tool (JVM-TI demo):
  `java -agentlib:hprof=cpu=sample com.example.Class`

JVM Profiling Tools

Java, Scala, Groovy, etc.

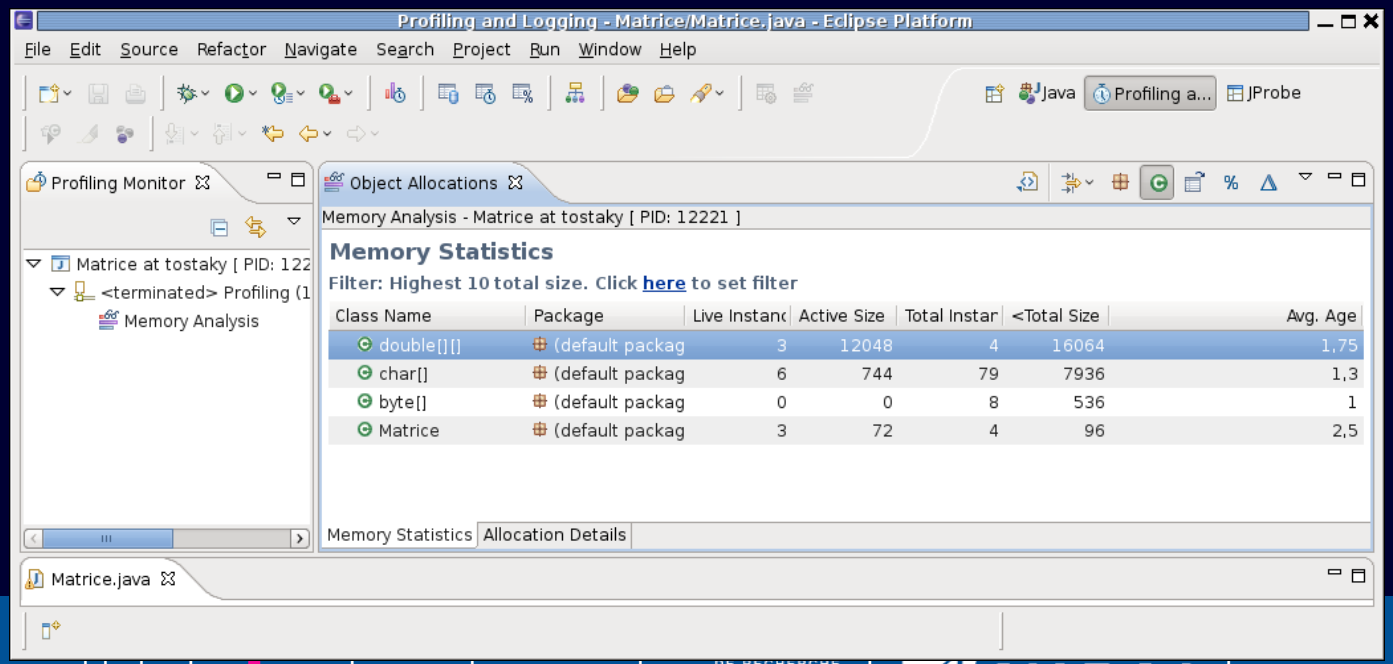
- **HProf**, built-in profiling tool (JVM-TI demo):
  `java -agentlib:hprof=cpu=sample com.example.Class`
- **JRat** (Java Runtime Analysis Toolkit),
  `http://jrat.sourceforge.net/`



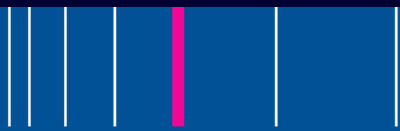
JVM Profiling Tools

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/`



Outline



Profilers for Other Languages

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

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

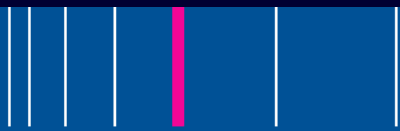
Why system-wide?

- 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 opcontrol --init
$ sudo opcontrol --start

$ run my favorite application(s)...
$ sudo opcontrol --stop
```

System-Wide Profiling with OProfile

http://oprofile.sourceforge.net/

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

$ run my favorite application(s)...
$ sudo opcontrol --stop

$ opreport -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

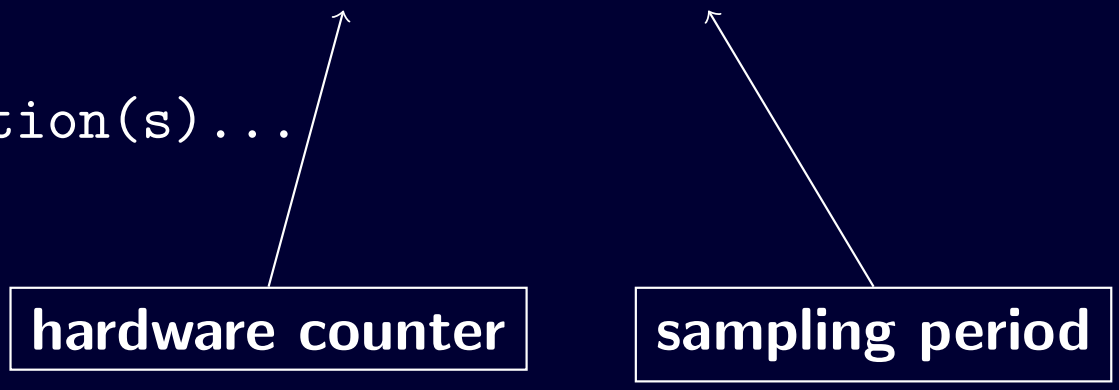
http://oprofile.sourceforge.net/

```
$ 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_progra
...
```

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
```



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, plafrim-support@inria.fr

Outline

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	≤ 1	≈ 3	≈ 14	≈ 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	≤ 1	≈ 3	≈ 14	≈ 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 $\Longrightarrow$ CPU pre-fetches 64 B
- Version 2 fetches 64 B for each access but uses only 4 B
- $\Longrightarrow$ 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

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?

http://sed.bordeaux.inria.fr/

sed-bordeaux@inria.fr

ludovic.courtes@inria.fr