# Profiling & Optimization

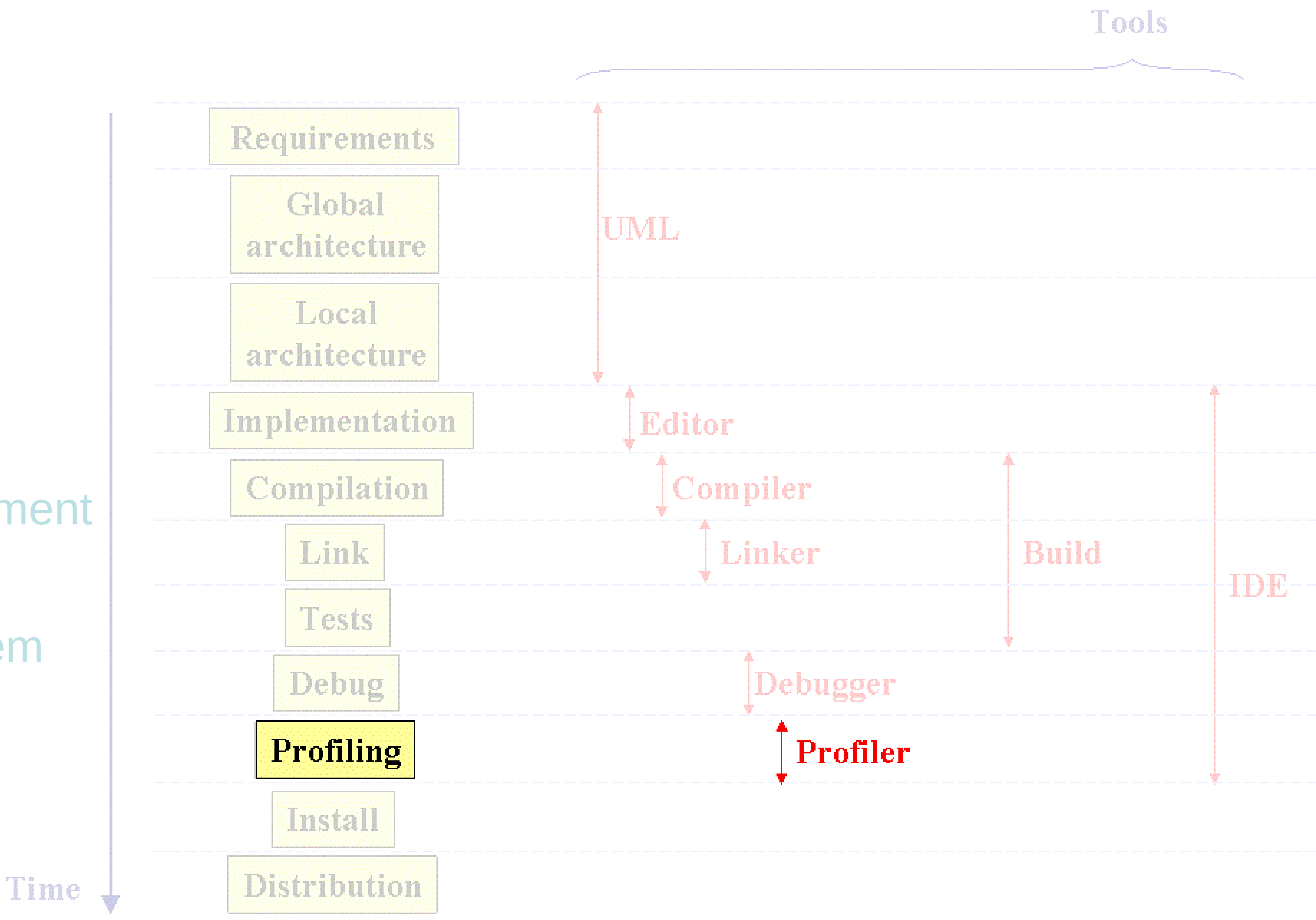
David Geldreich (DREAM)

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA
SOPHIA ANTIPOLIS

# Profiling & Optimization

- Introduction
- Analysis/Design
- Build
- Tests
- Debug
- **Profiling**
- Project management
- Documentation
- Versioning system
- IDE
- GForge
- Conclusion



# Outline

- **Profiling**
- Tools
- Optimization

# Profiling

No optimization without profiling

Not everyone has a P4 @ 3Ghz

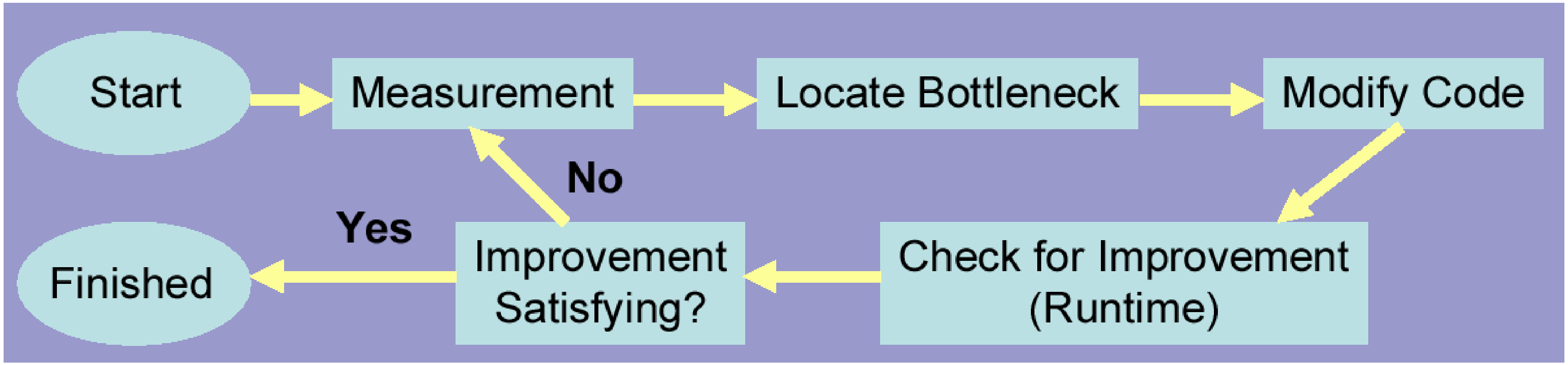
“*We should forget about small efficiencies, say about 97% of the time : premature optimization is the root of all evil.*” Donald Knuth

# Profiling : When ?

- To choose among several algorithms for a given problem
- To check the awaited behavior at runtime
- To find the parts of the code to be optimized

# Profiling : How ?

- On fully implemented (and also tested) code
- On a “release” version (optimized by the compiler, …)
- On representative data
- The optimization cycle :



# What do we measure ?

- Understand what’s going on :
  - OS: scheduling, memory management, hard drives, network
  - Compiler : optimization
  - CPU architecture, chipset, memory
  - Libraries used
- If an application is limited by its I/O, useless to improve the calculation part.
- Here we’ll limit ourselves to CPU & memory performance.

# Measurement methods

- Manual
- Source instrumentation
- Statistical measure (sampling)
- Simulation
- Hardware counters

# Outline

- Profiling
- **Tools**
- Optimization

# Tools

| | Manual | Instr. | Sampling | Simulation | Hardware Counter |
|---|---|---|---|---|---|
| ▪ system timers | X | | | | |
| ▪ gprof / gcc –pg | | X | X | | |
| ▪ valgrind(callgrind)/kcachegrind | | | | X | |
| IBM Rational quantify | | X | X | | |
| Oprofile | | | X | | |
| Intel Vtune | | | X | | X |
| PAPI (Performance API) | | | | | X |
| JVMPI (Java Virtual Machine Profiler)<br>• ▪ runhprof,<br>• ▪ Eclipse TPTP (Test&Performance)<br>• OptimizeIT, JProbe, JMP (Memory Profiler) | | | X | | |
| Shark (MacOS X) | | | X | | X |

# Tools

| | Manual | Instr. | Sampling | Simulation | Hardware Counter |
|---|---|---|---|---|---|
| **system timers** | **X** | | | | |
| gprof / gcc -pg | | X | X | | |
| valgrind(callgrind)/kcachegrind | | | | X | |
| IBM Rational quantify | | X | X | | |
| Oprofile | | | X | | |
| Intel Vtune | | | X | | X |
| PAPI (Performance API) | | | | | X |
| JVMPI (Java Virtual Machine Profiler)<br>• runhprof,<br>• Eclipse TPTP (Test&Performance)<br>• OptimizeIT, JProbe, JMP (Memory Profiler) | | | X | | |
| Shark (MacOS X) | | | X | | X |

# Using the tools : system timers

- You have to **know the timer's resolution**
- Windows :
  - `QueryPerformanceCounter()/QueryPerformanceFrequency()`
- Linux/Unix :
  - `gettimeofday()`
  - `clock()`
- Java :
  - `System.currentTimeMillis()`
  - `System.currentTimeNanos()`
- Intel CPU counter : RDTSC (ReaD Time Stamp Counter)

# Tools

| | Manual | Instr. | Sampling | Simulation | Hardware Counter |
|---|---|---|---|---|---|
| system timers | X | | | | |
| **gprof / gcc -pg** | | **X** | **X** | | |
| valgrind(callgrind)/kcachegrind | | | | X | |
| IBM Rational quantify | | X | X | | |
| Oprofile | | | X | | |
| Intel Vtune | | | X | | X |
| PAPI (Performance API) | | | | | X |
| JVMPI (Java Virtual Machine Profiler)<br>• runhprof,<br>• Eclipse TPTP (Test&Performance)<br>• OptimizeIT, JProbe, JMP (Memory Profiler) | | | X | | |
| Shark (MacOS X) | | | X | | X |

# Using the tools : gprof

gprof (compiler generated instrumentation) :

- instrumentation : count the function calls
- temporal sampling
- compile with `gcc -pg`
- create `gmon.out` file at runtime

Drawbacks

- line information not precise
- needs a complete recompilation
- results not always easy to analyze for large software

# Using the tools : gprof

dgeld@enesco:~/src/versions/dgeld/cours/optimisation

```
enesco$ make timapg
g++ -O3 -g -march=pentium4 -pg -o timapg tima.cpp image.cpp image2.cpp timer.cpp
enesco$ ./timapg
invert  : 304.63        911650545 ticks
invert2 : 107.179 ms    320743665 ticks
invert3 : 80.948 ms     242239312 ticks
invert4 : 63.167 ms     189028665 ticks
invert4b : 63.115 ms    188874075 ticks
invert5 : 63.17 ms      189038527 ticks
invert6 : 14.781 ms     44223030 ticks
enesco$
```

INRIA SOPHIA ANTIPOLIS

# Using the tools : gprof

```
dgeld@enesco:~/src/versions/dgeld/cours/optimisation
enesco$ gprof ./timapg  | head
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 45.00      0.27     0.27        1   270.00   270.00  image::invert()
 15.00      0.36     0.09        1    90.00    90.00  image::invert2()
 11.67      0.43     0.07        1    70.00    70.00  image::invert3()
 10.00      0.49     0.06        1    60.00    60.00  image::invert4b()
  8.33      0.54     0.05        1    50.00    50.00  image::invert4()
enesco$ gprof -l ./timapg  | head
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 43.33      0.26     0.26                             image::invert() (image.cpp:77 @ 8048e3d)
  9.17      0.32     0.06                             image::invert2() (image.cpp:88 @ 8048e7c)
  8.33      0.36     0.05                             image::invert5() (image.cpp:132 @ 8048f35)
  6.67      0.41     0.04                             image::invert4() (image.cpp:108 @ 8048ee1)
  5.83      0.44     0.04                             image::invert3() (image.cpp:98 @ 8048eb2)
enesco$
```

# Tools

| | Manual | Instr. | Sampling | Simulation | Hardware Counter |
|---|---|---|---|---|---|
| system timers | X | | | | |
| gprof / gcc -pg | | X | X | | |
| **valgrind(callgrind)/kcachegrind** | | | | **X** | |
| IBM Rational quantify | | X | X | | |
| Oprofile | | | X | | |
| Intel Vtune | | | X | | X |
| PAPI (Performance API) | | | | | X |
| JVMPI (Java Virtual Machine Profiler)<br>• runhprof,<br>• Eclipse TPTP (Test&Performance)<br>• OptimizeIT, JProbe, JMP (Memory Profiler) | | | X | | |
| Shark (MacOS X) | | | X | | X |

INRIA SOPHIA ANTIPOLIS

# Using the tools : callgrind

callgrind/kcachegrind : http://kcachegrind.sf.net/cgi-bin/show.cgi

- cache simulator on top of valgrind
- CPU simulation : estimate CPU cycles for each line of code
- analyze the data more easily with kcachegrind

Drawbacks

- time estimates can be inaccurate
- measure only the user part of the code
- analyzed software is 20-100 times slower, uses huge amount of memory.

# Using the tools : callgrind

callgrind/kcachegrind usage :

- on already compiled software :

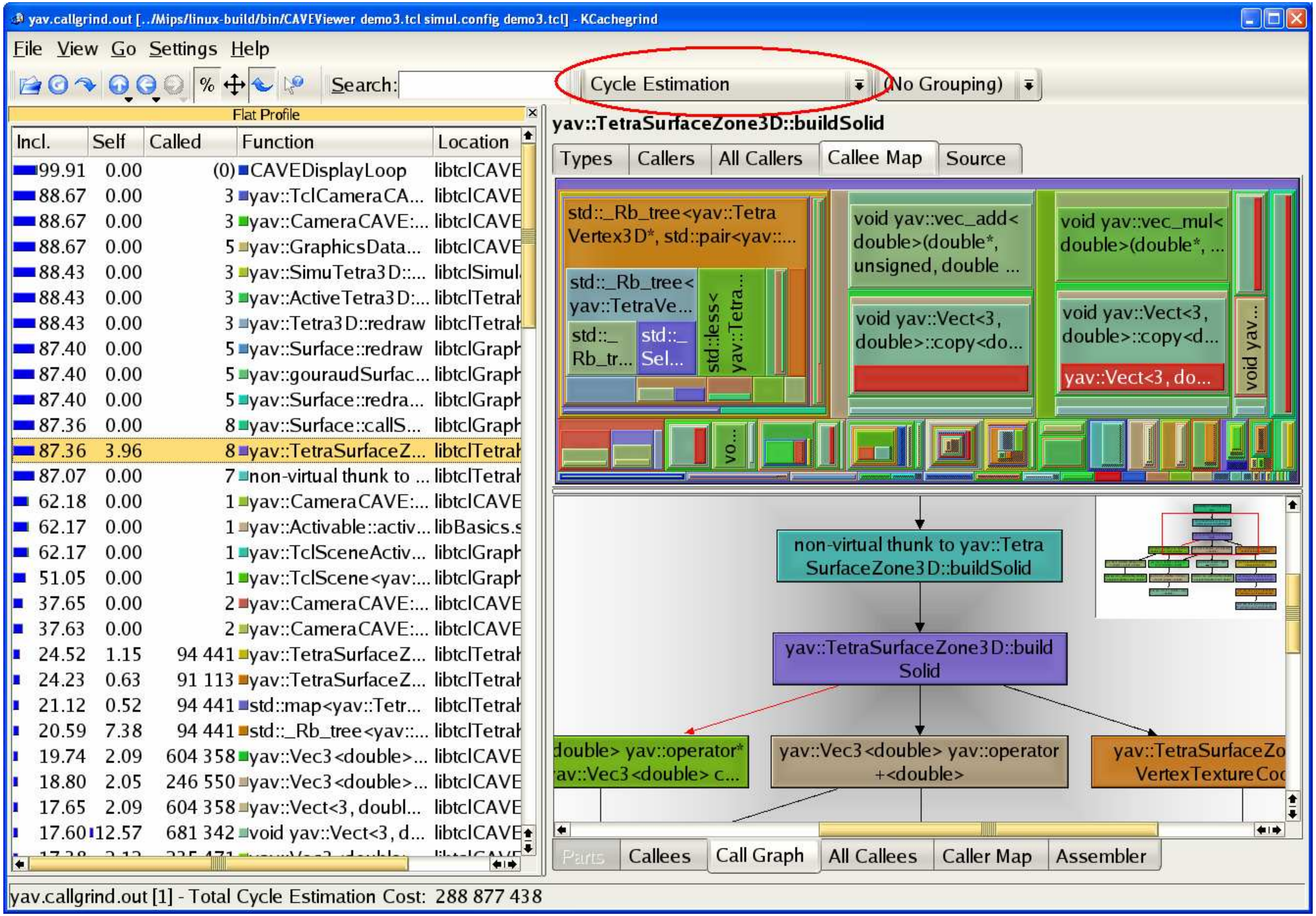
  `valgrind --tool=callgrind prog`
- generates `callgrind.out.xxx`
- analyzed with `callgrind_annotate` or `kcachegrind`
- to be usable in spite of its slowness :

  do not simulate cache usage : `--simulate-cache=no`

  start instrumentation only when needed :

  `--instr-atstart=no / callgrind_control -i on`

# Using the tools : callgrind



# Using the tools : massif

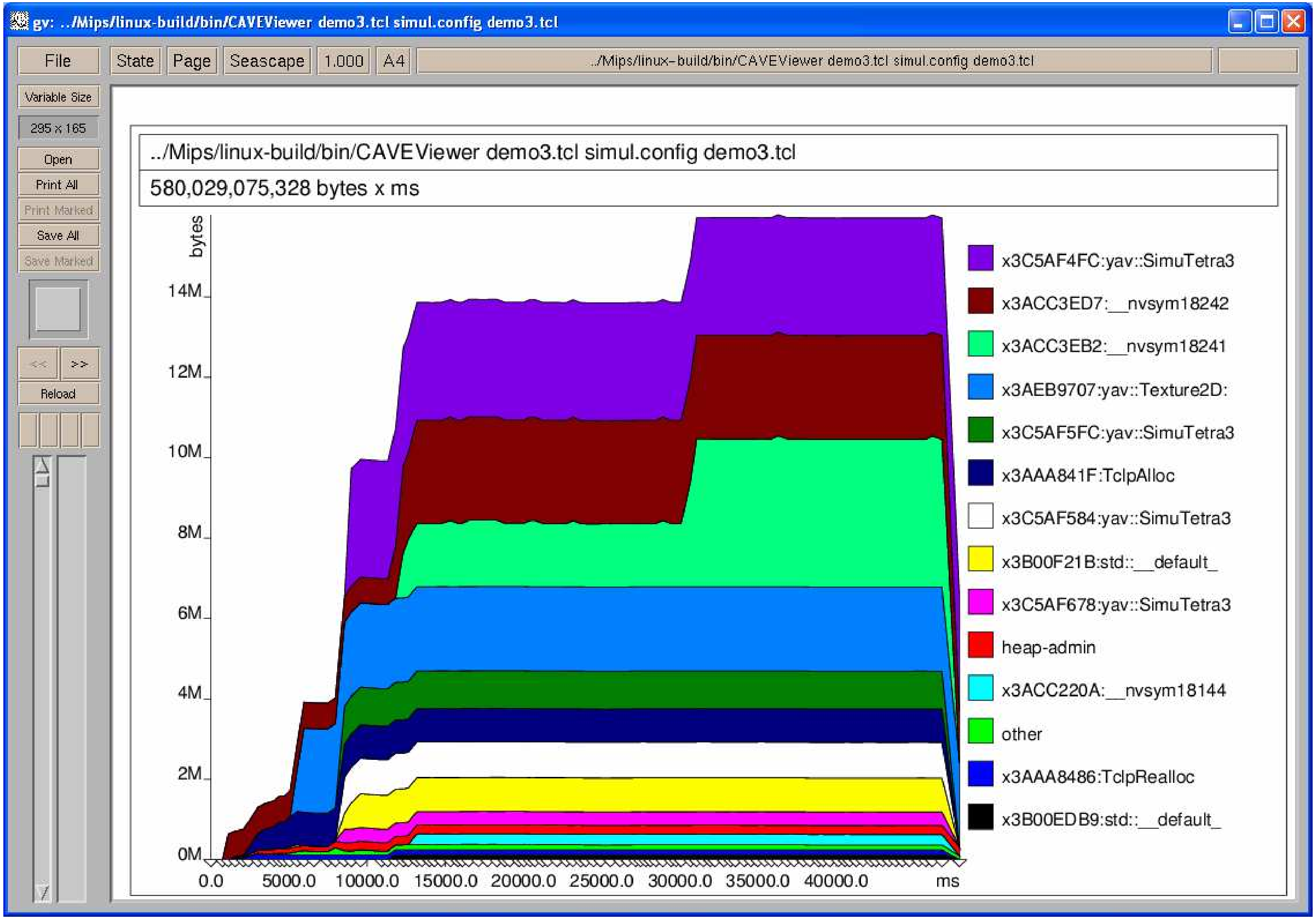
massif (heap profiler) : http://valgrind.org/info/tools.html#massif

- another valgrind tool to be used on compiled software :

```
valgrind --tool=massif prog
```

- generates `massif.xxx.ps` : memory usage vs. time
- `massif.xxx.txt` : which part of code uses what

# Using the tools : massif



# Tools

| | Manual | Instr. | Sampling | Simulation | Hardware Counter |
|---|---|---|---|---|---|
| system timers | X | | | | |
| gprof / gcc -pg | | X | X | | |
| valgrind(callgrind)/kcachegrind | | | | X | |
| IBM Rational quantify | | X | X | | |
| Oprofile | | | X | | |
| Intel Vtune | | | X | | X |
| PAPI (Performance API) | | | | | X |
| **JVMPI (JVM Profiler)**<br>• **runhprof,**<br>• **Eclipse TPTP (Test&Performance)** | | | **X** | | |
| Shark (MacOS X) | | | X | | X |

# Using the tools : runhprof

java/runhprof

- SUN's JVM extension

```
java -Xrunhprof:cpu=samples,depth=6,thread=y prog
```

- generates `java.hprof.txt` file
- analyzed with perfanal :

```
java -jar PerfAnal.jar java.hprof.txt
```

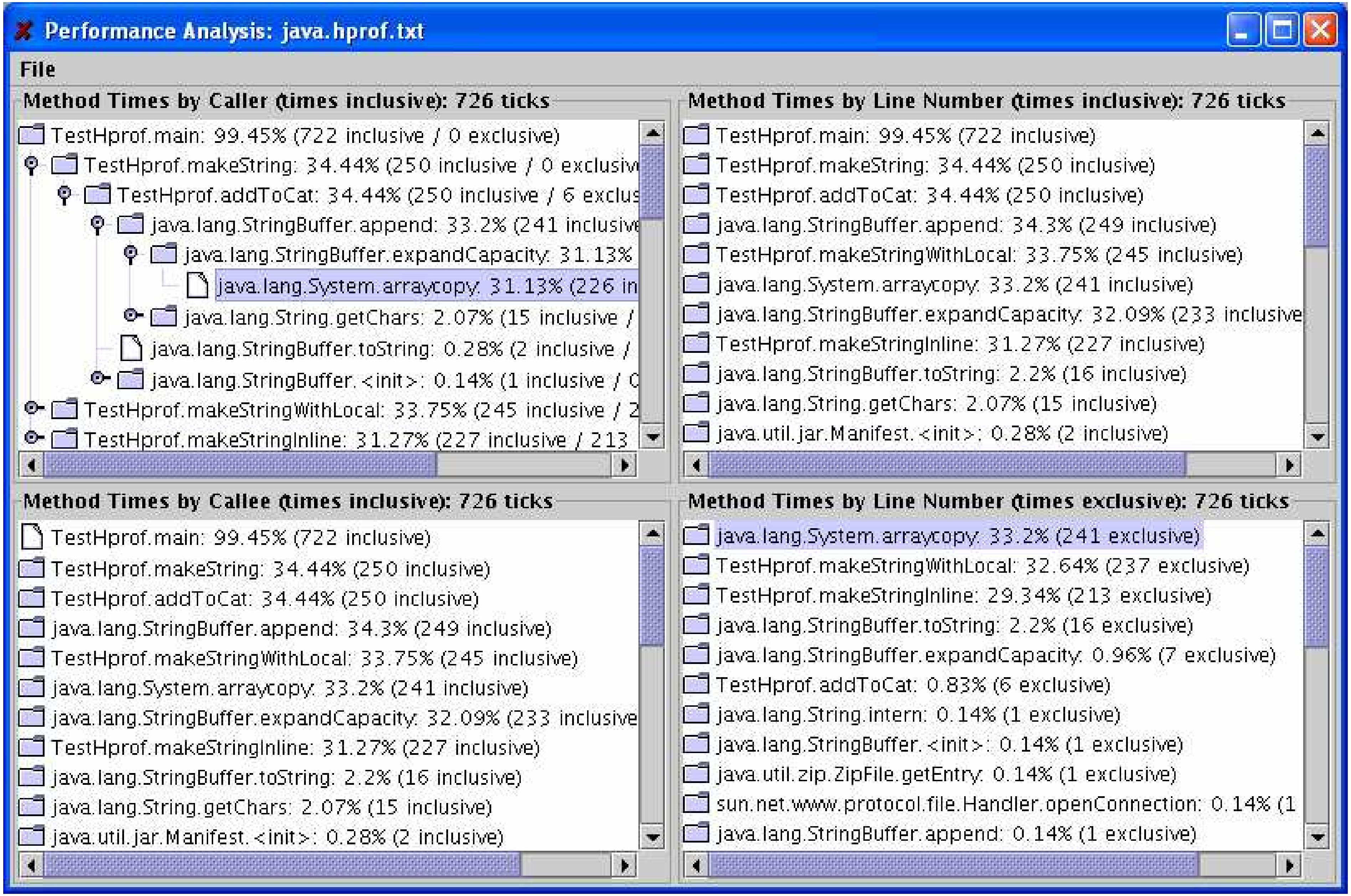
- memory :

```
java -Xrunhprof:heap=all prog
```

Drawbacks

- coarse sampling
- does not use all the possibilities of JVMPI

# Using the tools : runhprof



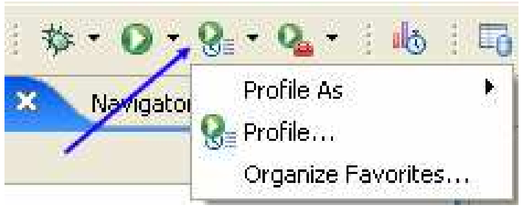
# Using the tools :
# Eclipse Test & Performance Tools Platform

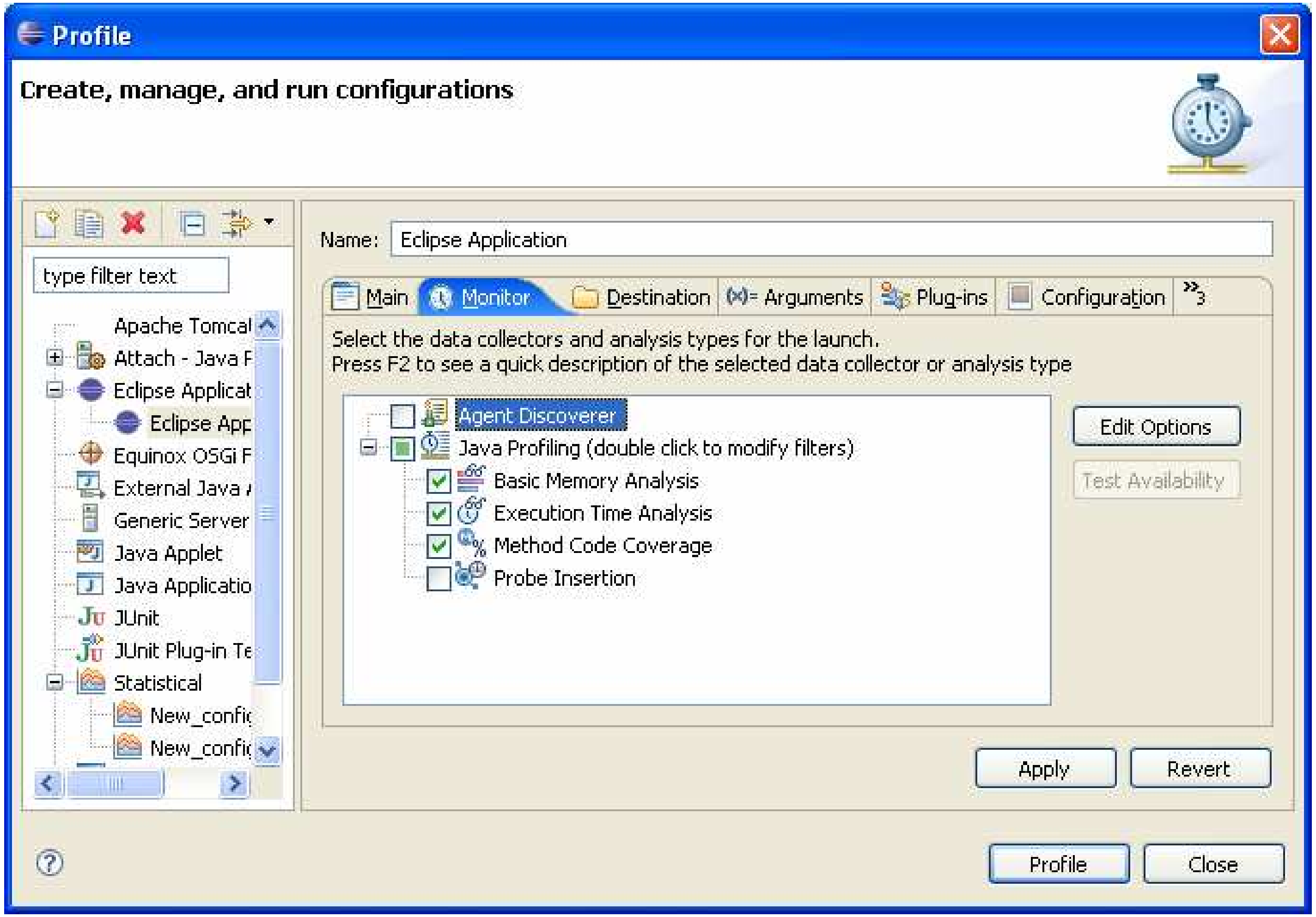
TPTP : tools integrated into Eclipse

Supports only Java (for the moment)

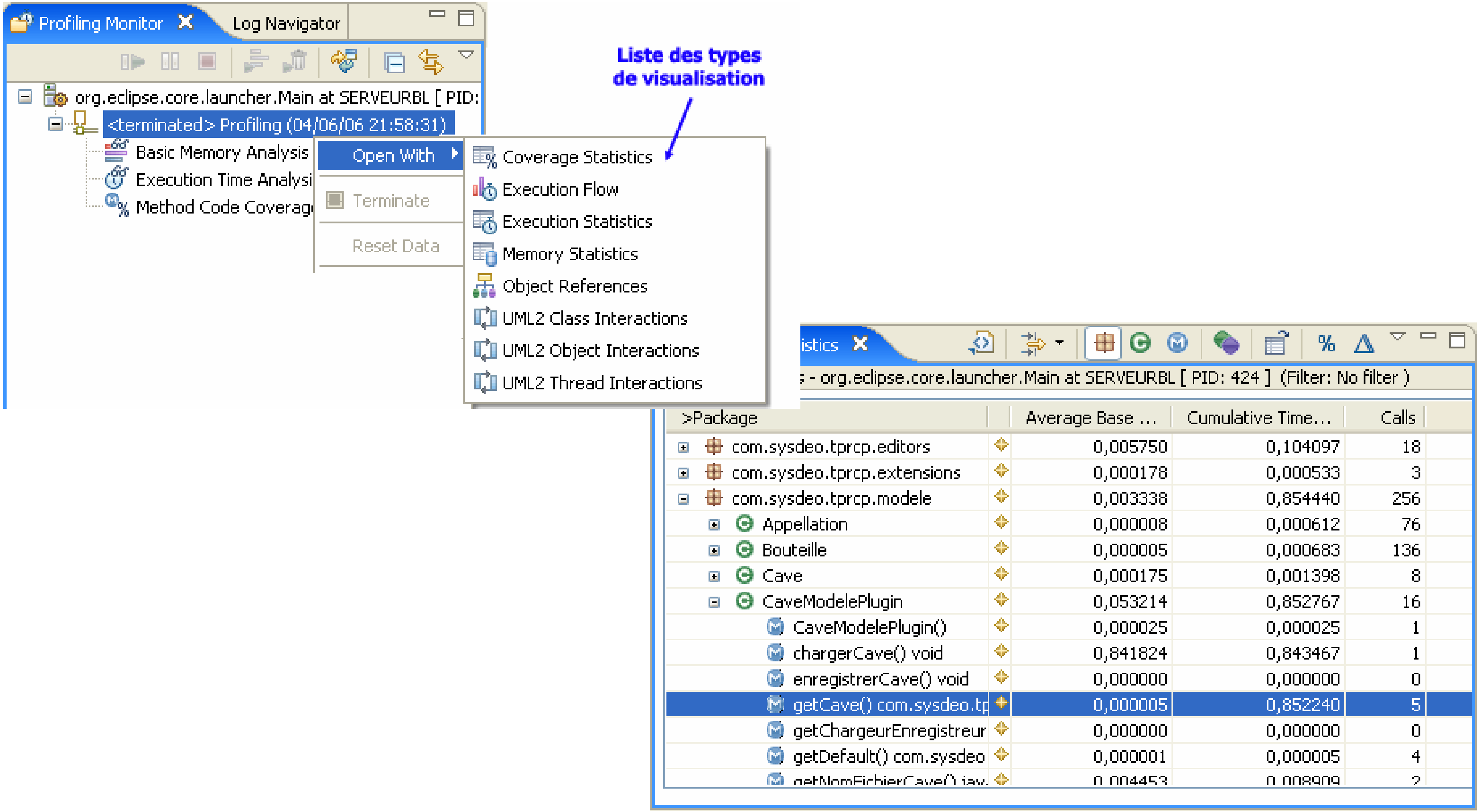
Profiling of local or distributed software

# Using the tools : Eclipse Test & Performance Tools Platform



# Using the tools : Eclipse Test & Performance Tools Platform



# Outline

- Profiling
- Tools
- **<u>Optimization</u>**

# Optimization

- No premature optimization
- Keep the code maintainable
- Do not over-optimize

# Optimization (continued)

- Find a better algorithm
  - the constant factor of the complexity can be significant
- Memory access : first cause of slowness
- Use already optimized libraries
- Limit the number of calls to expensive functions
- Write performance benchmarks/tests
  - allows one to check that the performance has not degraded
- The bottleneck moves at each optimization step
  - example : I/O can become blocking

# Optimization example

- Optimization example : image inversion (5000x5000)

```
for (int x = 0; x < w; x++)
    for (int y = 0; y < h; y++)
        data[y][x] = 255 - data[y][x];
```

| | |
|---|---|
| Without compiler optimization : | 435 ms |
| Compiler optimization (-O3) : | 316 ms |
| Improve memory access locality : | 107 ms |
| Suppress double dereference on `data` : | 94 ms |
| Constant code outside of the loop : | 63 ms (~7x) |
| OpenMP parallelization : | 38 ms |
| Using MMX assembly : | 26 ms |

# Conclusion

- No premature optimization
- Know your profiling tool
- Keep the code maintainable
- Do not over-optimize

# Questions ?