Performance Analysis and Optimization
MAQAO Tool

Andrés S. CHARIF-RUBIAL
achar@exascale-computing.eu
Performance Evaluation Tools Team

CERN 2nd P.T. Workshop - 21 November 2013

The Lab

- Joint Lab
- Research axes
  - Performance evaluation
  - Application characterization
  - Energy
- We work with ISV partners
- Feedback from our user community

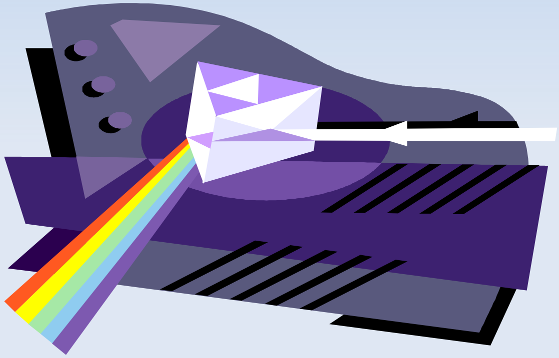
Outline

- Introduction
- MAQAO Toolchain
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules
  - Memory behavior characterization
  - Dynamic bottleneck Analyzer
  - Value profiler
  - PAMDA Methodology
  - Binary Instrumentation Language
- Conclusion

Introduction

Performance analysis

- Understand the performance of an application
  - How well it behaves on a given machine



- What are the issues ?
- Generally a multifaceted problem
  - Maximizing the number of views = better understand
- Use techniques and tools to understand issues
- Once understood ⇨ Optimize application

Introduction

Compilation toolchain

- Compiler remains your best friend
  - Be sure to select proper flags (e.g., -xavx)
  - Pragmas: Unrolling, Vector alignment
  - O2 V.S. O3
  - Vectorisation/optimisation report

MAQAO Tool

General information

- Open source (LGPL 3.0)
  - Currently binary release
  - Source release by mid December
  - Available tutorials

www.maqao.org

- Available for x86-64 and Xeon Phi
  - ARM version being developped @ University of Bordeaux

MAQAO Tool

General information

- Audience
  - User/Tool developer: analysis and optimisation tool
  - Performance tool developer: framework services
    - TAU: tau_rewrite (MIL)
    - ScoreP Framework: on-going effort (MIL)

Binary Instrumentation for Scalable Performance Measurement of OpenMP Applications. In *International Conference on Parallel Computing*, Parco2013, Munich, Germany, September 2013.

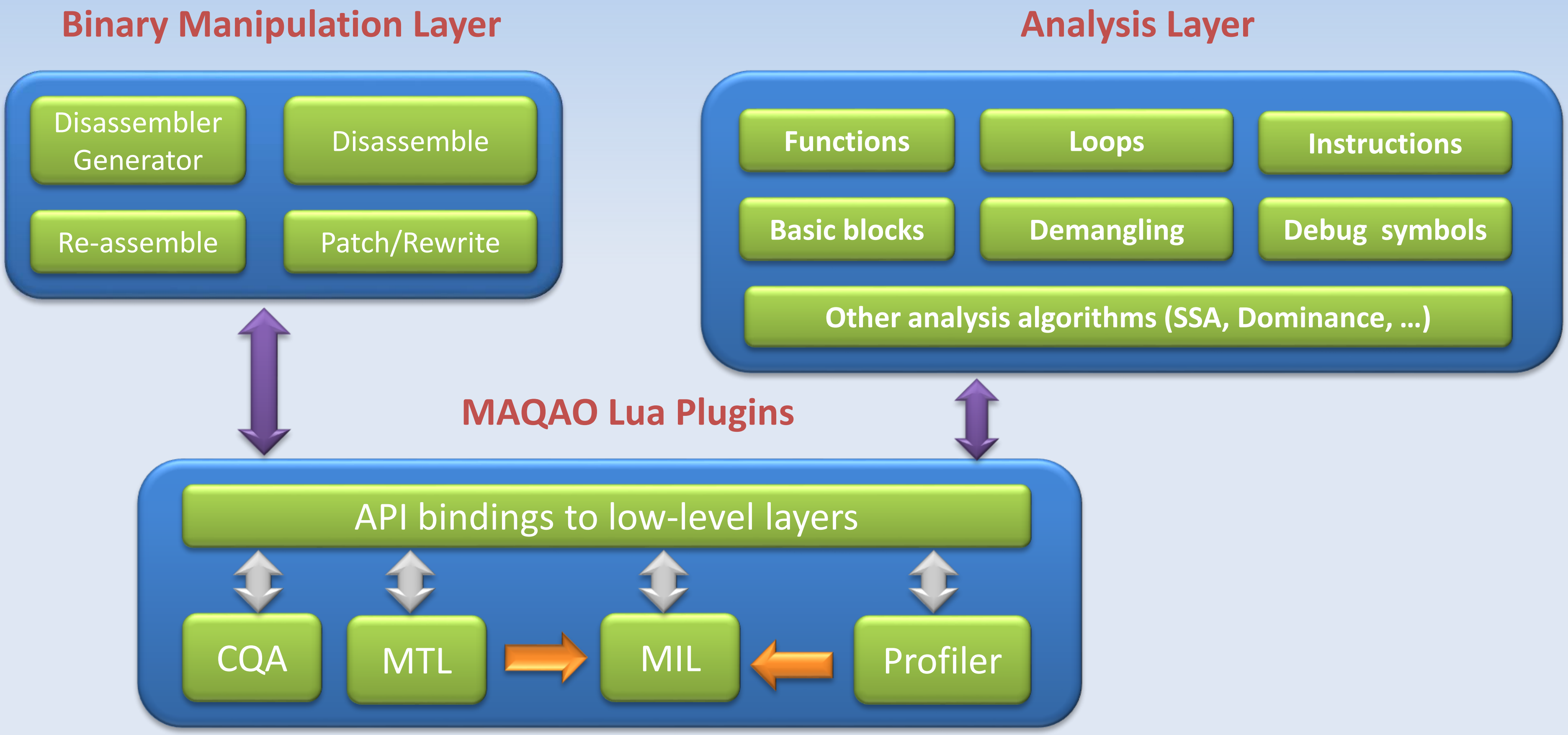
  - Research
- Easy install/embeed

Outline

- Introduction
- MAQAO Toolchain
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules
  - Memory behavior characterization
  - Dynamic bottleneck Analyzer
  - Value profiler
  - PAMDA Methodology
  - Binary Instrumentation Language
- Conclusion

MAQAO Tool

Framework overview



MAQAO Tool

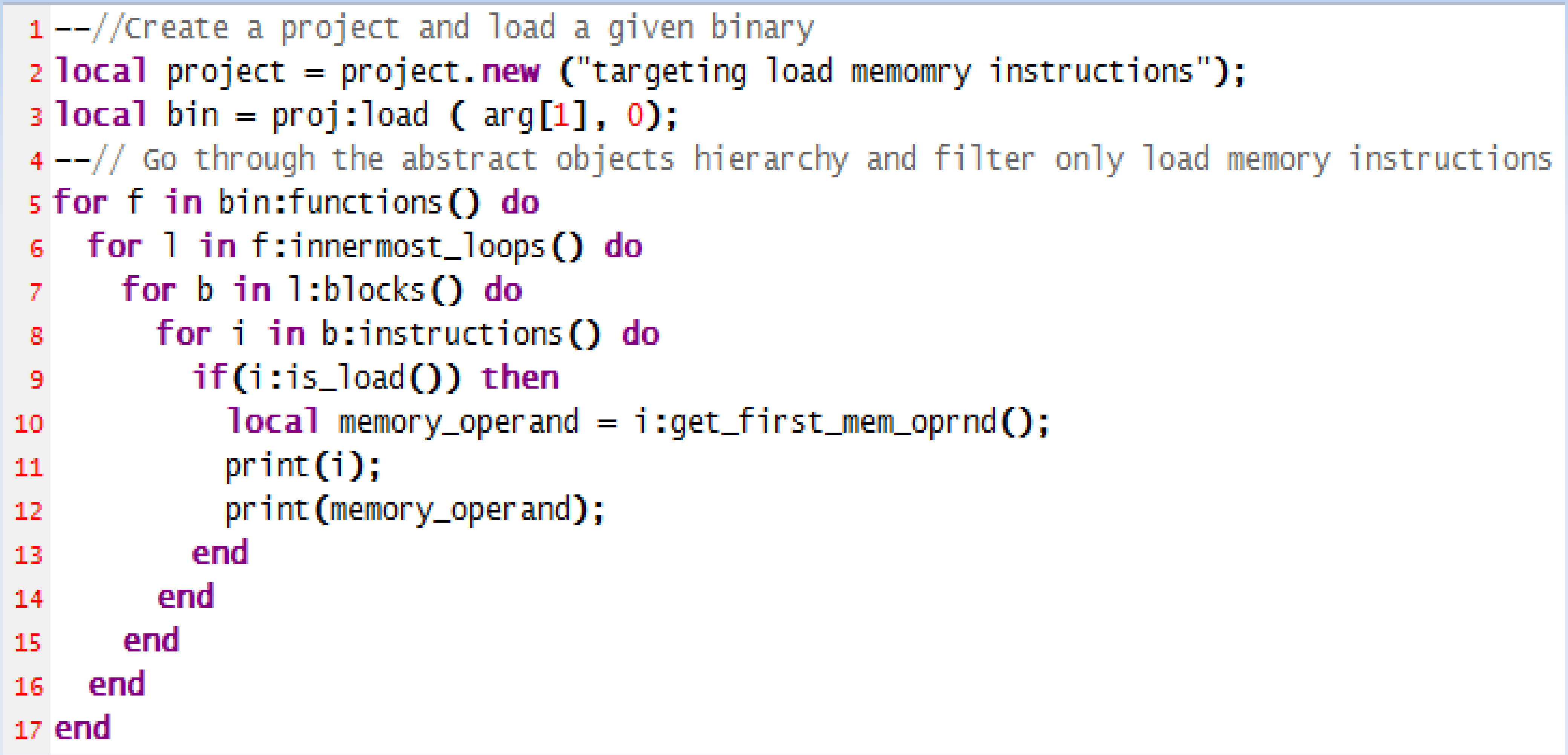
Framework overview

- Scripting language
  - Lua language : simplicity and productivity
  - Fast prototyping
  - MAQAO Lua API : Access to services

MAQAO Tool

Framework overview

Example of script : Display memory instructions



```
--//Create a project and load a given binary
local project = project.new ("targeting load memomry instructions");
local bin = proj:load ( arg[1], 0);
--// Go through the abstract objects hierarchy and filter only load memory instructions
for f in bin:functions() do
  for l in f:innermost_loops() do
    for b in l:blocks() do
      for i in b:instructions() do
        if(i:is_load()) then
          local memory_operand = i:get_first_mem_oprnd();
          print(i);
          print(memory_operand);
        end
      end
    end
  end
end
```

MAQAO Tool

Analysis & Optimization

- Performance on one node (except Profiler)
- More precisely: core level
- Built on top of the Framework
- Target HPC codes: Loop-centric approach
- Precise methodology
- Produce high level reports
    - We deal with low level details
    - You get high level reports

Outline

- Introduction
- MAQAO Toolchain
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules
  - Memory behavior characterization
  - Dynamic bottleneck Analyzer
  - Value profiler
  - PAMDA Methodology
  - Binary Instrumentation Language

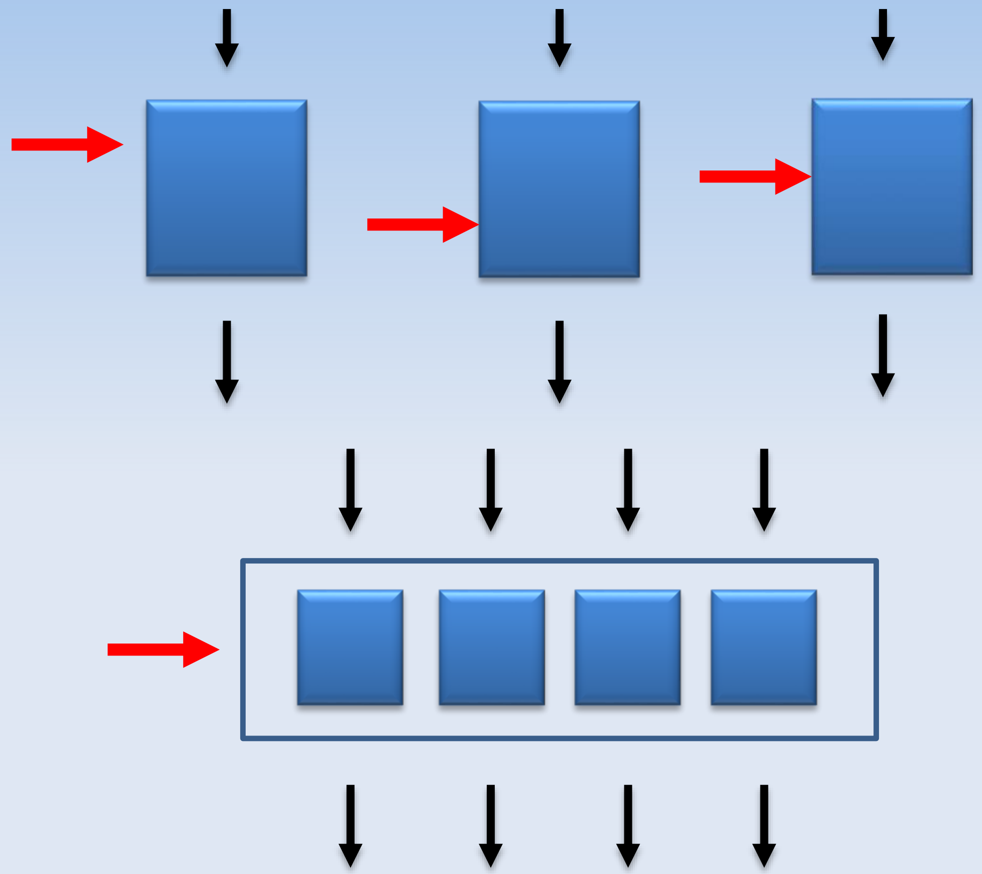
Pinpointing hotspots

Measurement methodology

- MAQAO Profiling
  - Instrumentation
    - Through binary rewriting
    - High overhead / More precision
  - Sampling
    - Hardware counter through perf_event_open system call
    - Very low overhead / less details

Pinpointing hotspots

Parallelism level



- SPMD
  - Program level
- SIMD
  - Instruction level
- By default MAQAO only considers system processes and threads

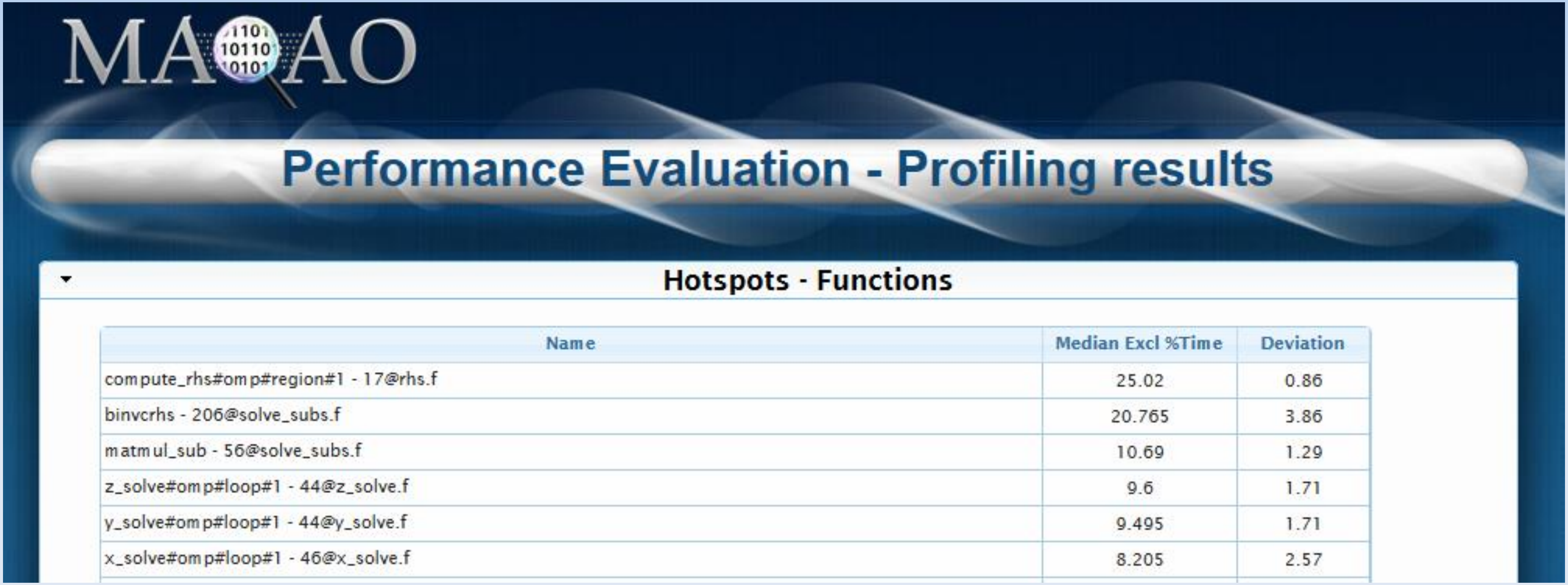
Pinpointing hotspots

Profiler

- Display functions and their exclusive time
  - Associated call chains and their contribution
  - Loops
- Innermost loops can then be analyzed by the code quality analyzer module (CQA)
- Command line and GUI (HTML) outputs

Pinpointing hotspots

GUI: High level hotspots vue

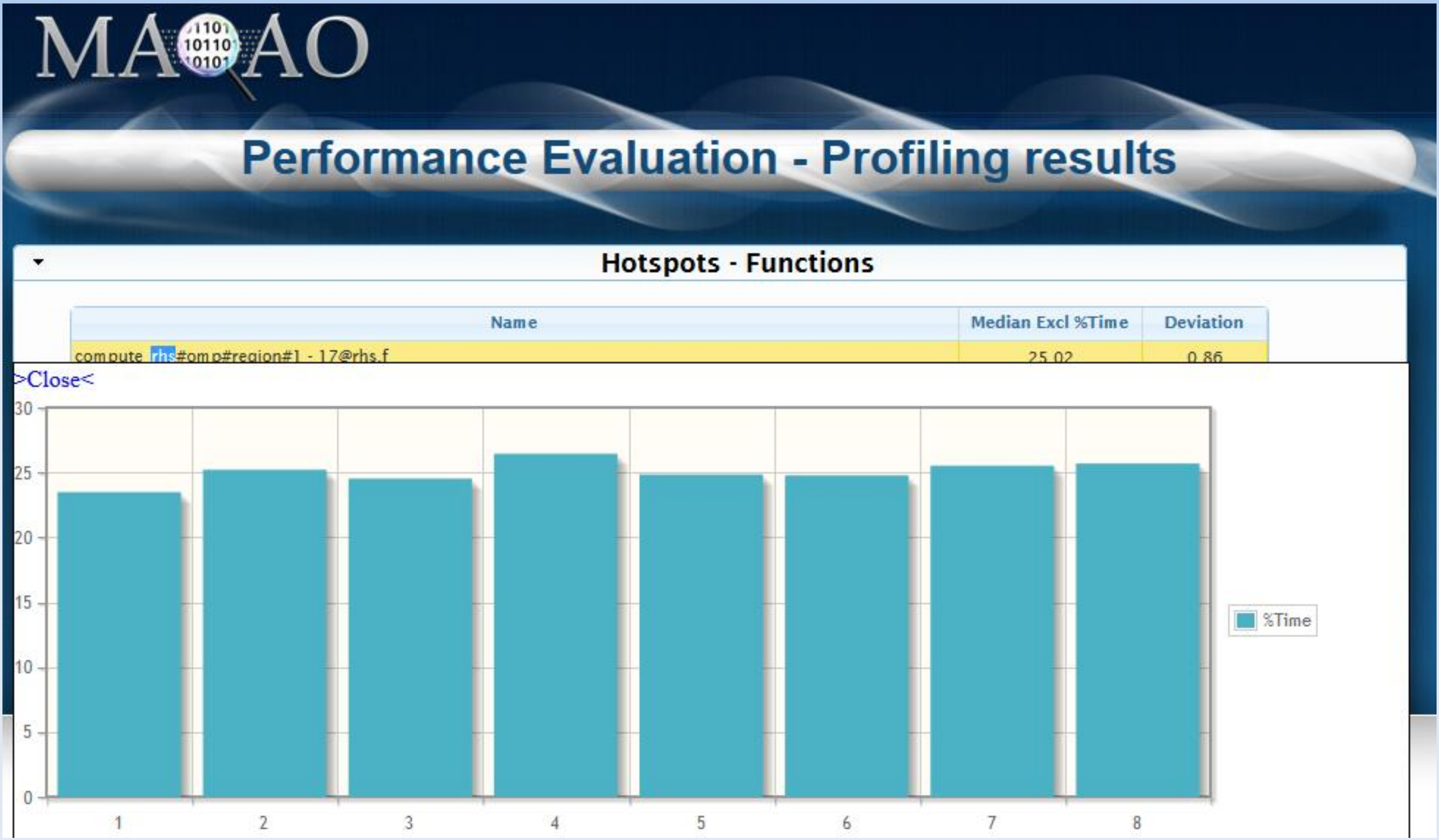


Hotspots - Functions

Name	Median Excl %Time	Deviation
compute_rhs#omp#region#1 - 17@rhs.f	25.02	0.86
binvcrhs - 206@solve_subs.f	20.765	3.86
matmul_sub - 56@solve_subs.f	10.69	1.29
z_solve#omp#loop#1 - 44@z_solve.f	9.6	1.71
y_solve#omp#loop#1 - 44@y_solve.f	9.495	1.71
x_solve#omp#loop#1 - 46@x_solve.f	8.205	2.57

Pinpointing hotspots

GUI: High level hotspots vue



Performance Evaluation - Profiling results

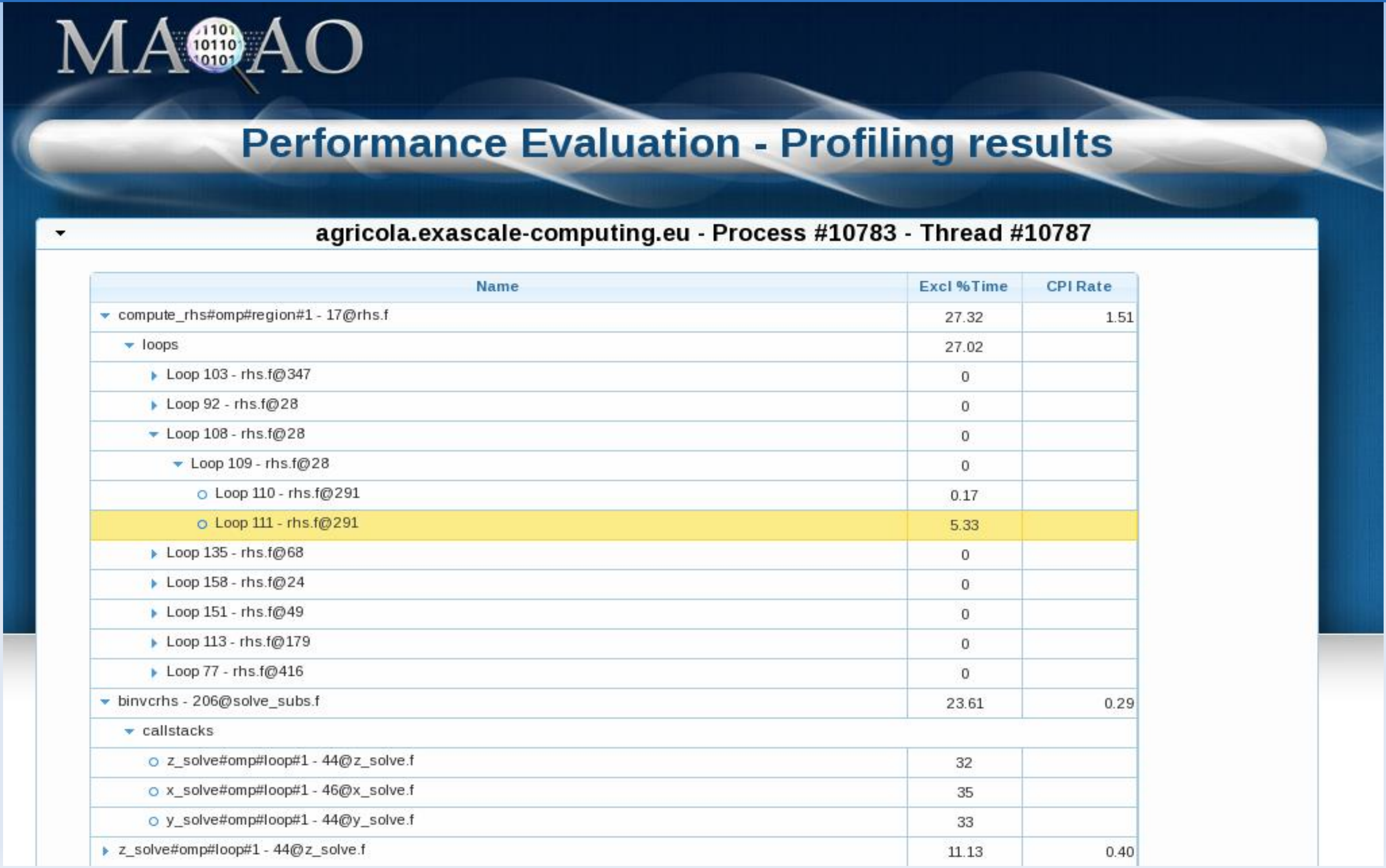
Hotspots - Functions

Name	Median Excl %Time	Deviation
compute rhs#omp#region#1 - 17@rhs.f	25.02	0.86

>Close<

Pinpointing hotspots

GUI: Node vue



Performance Evaluation - Profiling results

agricola.exascale-computing.eu - Process #10783 - Thread #10787

Name	Excl %Time	CPI Rate
compute_rhs#omp#region#1 - 17@rhs.f	27.32	1.51
loops	27.02	
Loop 103 - rhs.f@347	0	
Loop 92 - rhs.f@28	0	
Loop 108 - rhs.f@28	0	
Loop 109 - rhs.f@28	0	
Loop 110 - rhs.f@291	0.17	
Loop 111 - rhs.f@291	5.33	
Loop 135 - rhs.f@68	0	
Loop 158 - rhs.f@24	0	
Loop 151 - rhs.f@49	0	
Loop 113 - rhs.f@179	0	
Loop 77 - rhs.f@416	0	
binvcrhs - 206@solve_subs.f	23.61	0.29
callstacks		
z_solve#omp#loop#1 - 44@z_solve.f	32	
x_solve#omp#loop#1 - 46@x_solve.f	35	
y_solve#omp#loop#1 - 44@y_solve.f	33	
z_solve#omp#loop#1 - 44@z_solve.f	11.13	0.40

Pinpointing hotspots

Detecting hotspots: function level

```
                                        Thread #14831
#########################################################################################
#                Function Name                |  Time % (Nb of Events )  |  CPI ratio   #
#########################################################################################
#  compute_rhs#omp#region#1 - 17@rhs.f         | 24.37 (21644)            | 2.97        #
#  binvcrhs - 206@solve_subs.f                 | 21.67 (19240)            | 0.58        #
#  matmul_sub - 56@solve_subs.f                | 10.29 (9136)             | 0.56        #
#  y_solve#omp#loop#1 - 42@y_solve.f           | 9.87 (8764)              | 0.78        #
#  z_solve#omp#loop#1 - 42@z_solve.f           | 9.44 (8384)              | 0.75        #
#  x_solve#omp#loop#1 - 44@x_solve.f           | 8.53 (7576)              | 0.67        #
#  __kmp_wait_sleep [libiomp5.so]              | 6.64 (5900)              | 1.24        #
#  matvec_sub - 5@solve_subs.f                 | 2.75 (2440)              | 0.67        #
#  __kmp_x86_pause [libiomp5.so]               | 2.13 (1892)              | 0.87        #
#  add#omp#loop#1 - 18@add.f                   | 2.07 (1840)              | 4.55        #
#  Others                                      | 1.06 (940)               | 1.81        #
#  __kmp_yield [libiomp5.so]                   | 0.40 (356)               | 44.50       #
#  lhsinit - 210@initialize.f                  | 0.29 (256)               | 2.29        #
#  binvrhs - 488@solve_subs.f                  | 0.16 (140)               | 0.85        #
#  exact_solution - 4@exact_solution.f         | 0.11 (96)                | 0.83        #
#  exact_rhs#omp#region#1 - 19@exact_rhs.f     | 0.06 (52)                | 1.18        #
#  initialize#omp#region#1 - 21@initialize.f   | 0.03 (28)                | 0.70        #
#  data.10538 [libc-2.12.so]                   | 0.02 (20)                | 1.25        #
#  __kmp_fork_call [libiomp5.so]               | 0.01 (8)                 | 0.00        #
#########################################################################################
```

Pinpointing hotspots

Detecting hotspots: loop level

```
                                                Thread #19689
########################################################################################################
#  Loop ID  |                  Source Infos                 |    Level    |  Time % (Nb of Events)  |  CPI ratio  #
########################################################################################################
#  191      |  z_solve#omp#loop#1 - 137,299@z_solve.f       |  Innermost  |  4.99 (132)             |  0.01       #
#  171      |  x_solve#omp#loop#1 - 137,299@x_solve.f       |  Innermost  |  4.99 (132)             |  0.01       #
#  182      |  y_solve#omp#loop#1 - 46,128@y_solve.f        |  Innermost  |  4.84 (128)             |  0.03       #
#  181      |  y_solve#omp#loop#1 - 136,298@y_solve.f       |  Innermost  |  4.24 (112)             |  0.01       #
#  192      |  z_solve#omp#loop#1 - 46,128@z_solve.f        |  Innermost  |  4.08 (108)             |  0.02       #
#  172      |  x_solve#omp#loop#1 - 48,130@x_solve.f        |  Innermost  |  3.63 (96)              |  0.02       #
#  111      |  compute_rhs#omp#region#1 - 291,336@rhs.f     |  Innermost  |  2.72 (72)              |  0.03       #
#  133      |  compute_rhs#omp#region#1 - 181,225@rhs.f     |  Innermost  |  2.57 (68)              |  0.03       #
#  149      |  compute_rhs#omp#region#1 - 70,119@rhs.f      |  Innermost  |  2.42 (64)              |  0.03       #
#  161      |  compute_rhs#omp#region#1 - 26,36@rhs.f       |  Innermost  |  2.27 (60)              |  0.12       #
#  198      |  add#omp#loop#1 - 22,23@add.f                 |  Innermost  |  2.27 (60)              |  0.09       #
#  96       |  compute_rhs#omp#region#1 - 375,379@rhs.f     |  Innermost  |  2.27 (60)              |  0.04       #
#  156      |  compute_rhs#omp#region#1 - 52,53@rhs.f       |  Innermost  |  2.12 (56)              |  0.10       #
#  186      |  z_solve#omp#loop#1 - 399,402@z_solve.f       |  InBetween  |  1.21 (32)              |  0.01       #
#  79       |  compute_rhs#omp#region#1 - 419,420@rhs.f     |  Innermost  |  1.06 (28)              |  0.06       #
#  176      |  y_solve#omp#loop#1 - 386,389@y_solve.f       |  InBetween  |  0.76 (20)              |  0.01       #
#  141      |  compute_rhs#omp#region#1 - 144,148@rhs.f     |  Innermost  |  0.76 (20)              |  0.01       #
#  121      |  compute_rhs#omp#region#1 - 252,256@rhs.f     |  Innermost  |  0.61 (16)              |  0.01       #
#  166      |  x_solve#omp#loop#1 - 387,390@x_solve.f       |  InBetween  |  0.61 (16)              |  0.00       #
#  189      |  z_solve#omp#loop#1 - 399,402@z_solve.f       |  Innermost  |  0.45 (12)              |  0.01       #
#  175      |  y_solve#omp#loop#1 - 386,389@y_solve.f       |  Innermost  |  0.45 (12)              |  0.01       #
#  190      |  z_solve#omp#loop#1 - 334,356@z_solve.f       |  Innermost  |  0.30 (8)               |  0.02       #
########################################################################################################
```

Pinpointing hotspots

Profiler

- Special features
  - User guided measuring (Ctrl-Z)
  - Counting mode combined with instrumentation
  - Define profile files for specific group of counters

- Underway features
  - Show source code / assembly correlation
  - Optionally add MPI/OpenMP rank
  - Add CQA module output directly at loop level

Outline

- Introduction
- MAQAO Toolchain
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules
  - Memory behavior characterization
  - Dynamic bottleneck Analyzer
  - Value profiler
  - PAMDA Methodology
  - Binary Instrumentation Language

Code Quality Analysis

Introduction

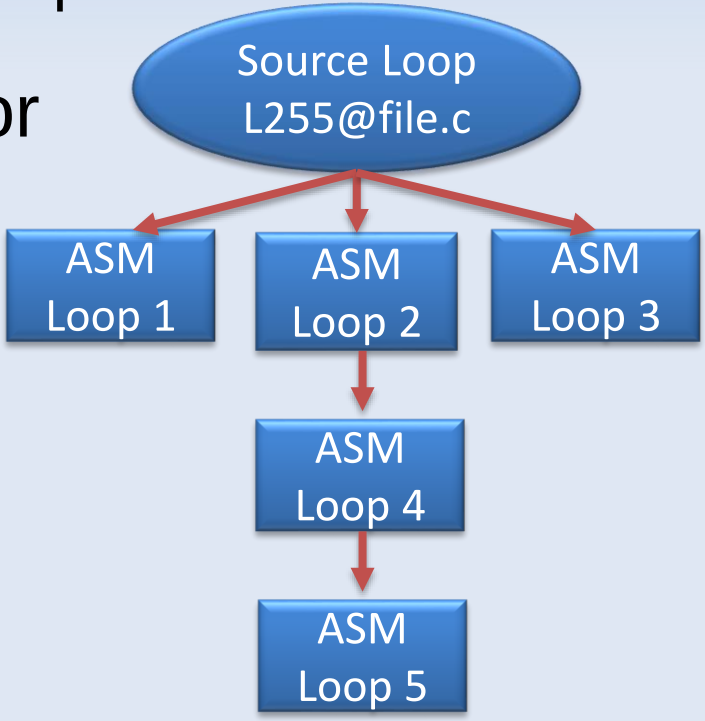
- Main performance issues:
    - Core level
    - Multicore interactions
    - Communications

- Most of the time core level is forgotten

Code Quality Analysis

Goals

- Static performance model
  - Targets innermost loops
    - source loop V.S. assembly loop
  - Take into account processor (micro)architecture
  - Assess code quality
    - Estimate performance
    - Degree of vectorization
    - Impact on micro architecture


Source Loop
L255@file.c
ASM Loop 1
ASM Loop 2
ASM Loop 3
ASM Loop 4
ASM Loop 5

Code Quality Analysis

Model

- Simulates the target (micro)architecture
  - Instructions description (latency, uops dispatch...) from our Microbenchmark module
  - Machine model (Intel documentation)
- For a given binary and micro-architecture, provides
  - Quality metrics (how well the binary is fitted to the micro architecture)
  - Static performance (lower bounds on cycles)
  - Hints and workarounds to improve static performance

Code Quality Analysis

Metrics

- Vectorization (ratio and speedup)
  - Allows to predict vectorization (if possible) speedup and increase vectorization ratio if it’s worth
- High latency instructions (division/square root)
  - Allows to use less precise but faster instructions like RCP (1/x) and RSQRT (1/sqrt(x))
- Unrolling (unroll factor detection)
  - Allows to statically predict performance for different unroll factors (find main loops)

Code Quality Analysis

Report example

```
Pathological cases
------------------
Your loop is processing FP elements but is NOT
OR PARTIALLY VECTORIZED.
Since your execution units are vector units,
only a fully vectorized loop can use their full
power.
By fully vectorizing your loop, you can lower
the cost of an iteration from 14.00 to 3.50
cycles (4.00x speedup).
Two propositions:
 - Try another compiler or update/tune your
current one:
  * gcc: use O3 or Ofast. If targeting IA32,
add mfpmath=sse combined with march=<cputype>,
msse or msse2.
  * icc: use the vec-report option to
understand why your loop was not vectorized. If
"existence of vector dependences", try the
IVDEP directive. If, using IVDEP,
"vectorization possible but seems inefficient",
try the VECTOR ALWAYS directive.
 - Remove inter-iterations dependences from
your loop and make it unit-stride.

WARNING: Fix as many pathological cases as you
can before reading the following sections.
```

```
Bottlenecks
-----------
The divide/square root unit is a bottleneck.
Try to reduce the number of division or square
root instructions.
If you accept to loose numerical precision, you
can speedup your code by passing the following
options to your compiler:
gcc: (ffast-math or Ofast) and mrecip
icc: this should be automatically done by
default

By removing all these bottlenecks, you can
lower the cost of an iteration from 14.00 to
1.50 cycles (9.33x speedup).
```

Outline

- Introduction
- MAQAO Toolchain
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules
  - Memory behavior characterization
  - Dynamic bottleneck Analyzer
  - Value profiler
  - PAMDA Methodology
  - Binary Instrumentation Language

Upcoming modules

- Dynamic bottleneck analyzer: differential analysis
- Memory characterization tool
  - Access patterns
  - Data reshaping
  - Cache simulator
- Value profiler
  - Function specialization / memorizing
  - Loops instances (iteration count) variations
- MPI & OpenMP scalable profiling

Outline

- Introduction
- MAQAO Toolchain
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules
  - Memory behavior characterization
  - Dynamic bottleneck Analyzer
  - Value profiler
  - PAMDA Methodology
  - Binary Instrumentation Language

Memory behavior characterization

- Single threaded aspects
  - Transformation opportunities, e.g.: loop interchange
  - Data reshaping opportunities , e.g.: array splitting
  - Detect alignment issues

Memory behavior characterization

Single threaded aspects: Inefficient patterns

Real code example: PNBENCH

```
for (int n=0; n<M; n++)
  if (lambdaz[n] > 0.) {
    for (int j=0; j<mesh.NCx; j++)
      for (int i=1; i<mesh.NCz; i++)
        J_upz[IDX3C(n,i,M,j,(mesh.NCz+1)*M)] = Jz[IDX3C(n,i-1,M,j,(mesh.NCz)*M)] * lambdaz[n];
  }
  if (lambdaz[n] < 0.){
```

MTL output

```
Load (Double) - Pattern: 8*i1  (Hits : 100% | Count : 1)
Load (Double) - Pattern: 8*i1+217600*i2+1088*i3  (Hits : 100% | Count : 1)
Store (Double)- Pattern: 8*i1+218688*i2+1088*i3  (Hits : 100% | Count : 1)
```

- Stride 1 (8/8) one access for outmost
- Poor access patterns for two instructions
- Idealy: smallest stides inside to outside
- Here: interchange **n** and **i** loops

Memory behavior characterization

Single threaded aspects: Inefficient patterns

Real code example: PNBENCH

- Example: **flux_numerique_z,** loop **193** (same for 195)
- Same kind of optimization for loops 204 and 206

Original

```
for (int n=0; n<M; n++) {
  if (lambdaz[n] > 0.){
    for (int j=0; j<NCx; j++)
      for (int i=1; i<NCz; i++) // loop 193
       J_upz[IDX3C(n,i,M,j,(NCz+1)*M)]=
Jz[IDX3C(n,i-1,M,j,(NCz)*M)] * lambdaz[n];
  }
  if (lambdaz[n] < 0.)
     …//loop 195
}
```

After transformation

```
for (int j=0; j<NCx; j++)
  for (int n=0; n<M; n++) {
    if (lambdax[n] > 0.){
      for (int i=1; i<NCz; i++) // loop 193
       J_upz[IDX3C(n,i,M,j,(NCz+1)*M)]=
Jz[IDX3C(n,i-1,M,j,(NCz)*M)] * lambdaz[n];
   }
   if (lambdaz[n] < 0.)
      …//loop 195
   }
```

7.7x local speedup (loops) → 1.4x GLOBAL speedup

Memory behavior characterization

Single threaded aspects: data alignment

- Instructions in original code not aligned:
  - Padding if complex structure
  - Compiler flags, pragmas to align (e.g.: vectors)
  - Allocate aligned memory: use posix_memalign()

- Architecture issue: even if aligned
  - Up to 10 cycles penalty
  - Micro benchmarking on each new machine
  - Warn user about values (alignment) to avoid

Outline

- Introduction
- MAQAO Toolchain
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules
  - Memory behavior characterization
  - Dynamic bottleneck Analyzer
  - Value profiler
  - PAMDA Methodology
  - Binary Instrumentation Language

Differential Analysis (DECAN)

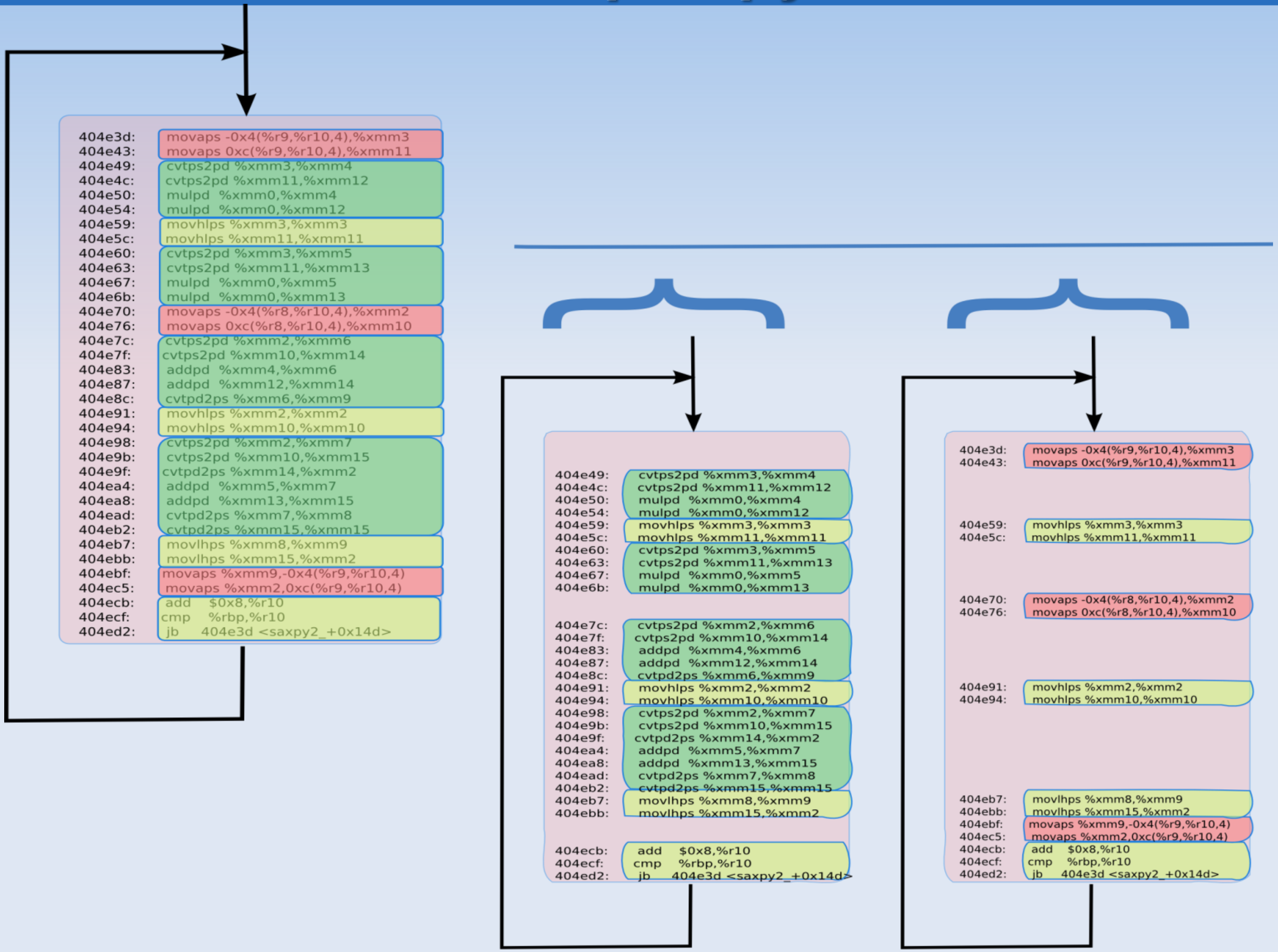
Principle & Usage

- Principle
  - Performance of the original loop is measured
  - Some instructions are removed in the loop body (for example loads and stores)
  - Performance of the transformed loop is measured
- Usage
  - Can perform sampling by transforming only 1 instance and abort execution
  - Can replay original loop execution after modified one
  - The Diff. Analysis speedup is an upper bound for optimization on the removed instructions

Quantifying performance bottleneck cost through differential analysis.
In *27th international ACM conference on International conference on supercomputing*, ICS'13

Differential Analysis (DECAN)

Loop copy



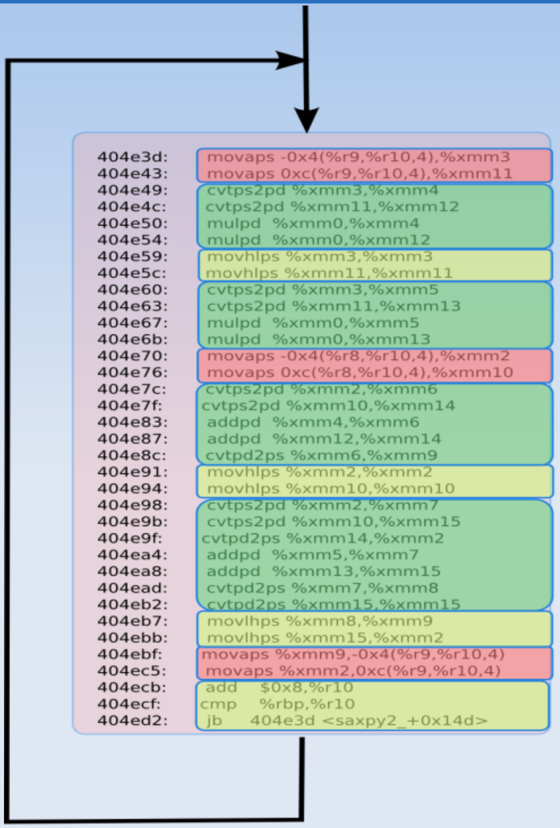
```
404e3d:  movaps -0x4(%r9,%r10,4),%xmm3
404e43:  movaps 0xc(%r9,%r10,4),%xmm11
404e49:  cvtps2pd %xmm3,%xmm4
404e4c:  cvtps2pd %xmm11,%xmm12
404e50:  mulpd  %xmm0,%xmm4
404e54:  mulpd  %xmm0,%xmm12
404e59:  movhlps %xmm3,%xmm3
404e5c:  movhlps %xmm11,%xmm11
404e60:  cvtps2pd %xmm3,%xmm5
404e63:  cvtps2pd %xmm11,%xmm13
404e67:  mulpd  %xmm0,%xmm5
404e6b:  mulpd  %xmm0,%xmm13
404e70:  movaps -0x4(%r8,%r10,4),%xmm2
404e76:  movaps 0xc(%r8,%r10,4),%xmm10
404e7c:  cvtps2pd %xmm2,%xmm6
404e7f:  cvtps2pd %xmm10,%xmm14
404e83:  addpd  %xmm4,%xmm6
404e87:  addpd  %xmm12,%xmm14
404e8c:  cvtpd2ps %xmm6,%xmm9
404e91:  movhlps %xmm2,%xmm2
404e94:  movhlps %xmm10,%xmm10
404e98:  cvtps2pd %xmm2,%xmm7
404e9b:  cvtps2pd %xmm10,%xmm15
404e9f:  cvtpd2ps %xmm14,%xmm2
404ea4:  addpd  %xmm5,%xmm7
404ea8:  addpd  %xmm13,%xmm15
404ead:  cvtpd2ps %xmm7,%xmm8
404eb2:  cvtpd2ps %xmm15,%xmm15
404eb7:  movlhps %xmm8,%xmm9
404ebb:  movlhps %xmm15,%xmm2
404ebf:  movaps %xmm9,-0x4(%r9,%r10,4)
404ec5:  movaps %xmm2,0xc(%r9,%r10,4)
404ecb:  add    $0x8,%r10
404ecf:  cmp    %rbp,%r10
404ed2:  jb     404e3d <saxpy2_+0x14d>
```

```
404e49:  cvtps2pd %xmm3,%xmm4
404e4c:  cvtps2pd %xmm11,%xmm12
404e50:  mulpd  %xmm0,%xmm4
404e54:  mulpd  %xmm0,%xmm12
404e59:  movhlps %xmm3,%xmm3
404e5c:  movhlps %xmm11,%xmm11
404e60:  cvtps2pd %xmm3,%xmm5
404e63:  cvtps2pd %xmm11,%xmm13
404e67:  mulpd  %xmm0,%xmm5
404e6b:  mulpd  %xmm0,%xmm13

404e7c:  cvtps2pd %xmm2,%xmm6
404e7f:  cvtps2pd %xmm10,%xmm14
404e83:  addpd  %xmm4,%xmm6
404e87:  addpd  %xmm12,%xmm14
404e8c:  cvtpd2ps %xmm6,%xmm9
404e91:  movhlps %xmm2,%xmm2
404e94:  movhlps %xmm10,%xmm10
404e98:  cvtps2pd %xmm2,%xmm7
404e9b:  cvtps2pd %xmm10,%xmm15
404e9f:  cvtpd2ps %xmm14,%xmm2
404ea4:  addpd  %xmm5,%xmm7
404ea8:  addpd  %xmm13,%xmm15
404ead:  cvtpd2ps %xmm7,%xmm8
404eb2:  cvtpd2ps %xmm15,%xmm15
404eb7:  movlhps %xmm8,%xmm9
404ebb:  movlhps %xmm15,%xmm2

404ecb:  add    $0x8,%r10
404ecf:  cmp    %rbp,%r10
404ed2:  jb     404e3d <saxpy2_+0x14d>
```

```
404e3d:  movaps -0x4(%r9,%r10,4),%xmm3
404e43:  movaps 0xc(%r9,%r10,4),%xmm11

404e59:  movhlps %xmm3,%xmm3
404e5c:  movhlps %xmm11,%xmm11

404e70:  movaps -0x4(%r8,%r10,4),%xmm2
404e76:  movaps 0xc(%r8,%r10,4),%xmm10

404e91:  movhlps %xmm2,%xmm2
404e94:  movhlps %xmm10,%xmm10

404eb7:  movlhps %xmm8,%xmm9
404ebb:  movlhps %xmm15,%xmm2
404ebf:  movaps %xmm9,-0x4(%r9,%r10,4)
404ec5:  movaps %xmm2,0xc(%r9,%r10,4)
404ecb:  add    $0x8,%r10
404ecf:  cmp    %rbp,%r10
404ed2:  jb     404e3d <saxpy2_+0x14d>
```

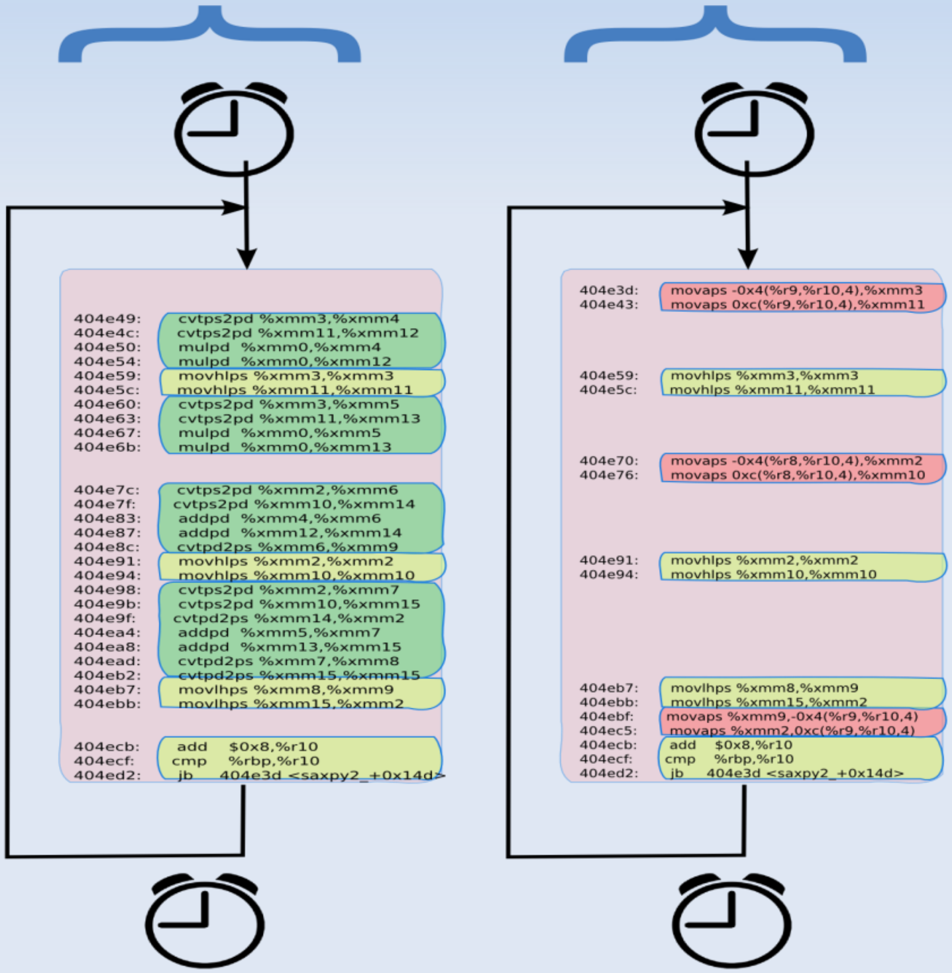
Differential Analysis (DECAN)

Timing



```
404e3d:  movaps -0x4(%r9,%r10,4),%xmm3
404e43:  movaps 0xc(%r9,%r10,4),%xmm11
404e49:  cvtps2pd %xmm3,%xmm4
404e4c:  cvtps2pd %xmm11,%xmm12
404e50:  mulpd  %xmm0,%xmm4
404e54:  mulpd  %xmm0,%xmm12
404e59:  movhlps %xmm3,%xmm3
404e5c:  movhlps %xmm11,%xmm11
404e60:  cvtps2pd %xmm3,%xmm5
404e63:  cvtps2pd %xmm11,%xmm13
404e67:  mulpd  %xmm0,%xmm5
404e6b:  mulpd  %xmm0,%xmm13
404e70:  movaps -0x4(%r8,%r10,4),%xmm2
404e76:  movaps 0xc(%r8,%r10,4),%xmm10
404e7c:  cvtps2pd %xmm2,%xmm6
404e7f:  cvtps2pd %xmm10,%xmm14
404e83:  addpd  %xmm4,%xmm6
404e87:  addpd  %xmm12,%xmm14
404e8c:  cvtpd2ps %xmm6,%xmm9
404e91:  movhlps %xmm2,%xmm2
404e94:  movhlps %xmm10,%xmm10
404e98:  cvtps2pd %xmm2,%xmm7
404e9b:  cvtps2pd %xmm10,%xmm15
404e9f:  cvtpd2ps %xmm14,%xmm2
404ea4:  addpd  %xmm5,%xmm7
404ea8:  addpd  %xmm13,%xmm15
404ead:  cvtpd2ps %xmm7,%xmm8
404eb2:  cvtpd2ps %xmm15,%xmm15
404eb7:  movlhps %xmm8,%xmm9
404ebb:  movlhps %xmm15,%xmm2
404ebf:  movaps %xmm9,-0x4(%r9,%r10,4)
404ec5:  movaps %xmm2,0xc(%r9,%r10,4)
404ecb:  add    $0x8,%r10
404ecf:  cmp    %rbp,%r10
404ed2:  jb     404e3d <saxpy2_+0x14d>
```

Monitor - Execution times
- Loop Iteration numbers
- hardware counter values



```
404e49:  cvtps2pd %xmm3,%xmm4
404e4c:  cvtps2pd %xmm11,%xmm12
404e50:  mulpd  %xmm0,%xmm4
404e54:  mulpd  %xmm0,%xmm12
404e59:  movhlps %xmm3,%xmm3
404e5c:  movhlps %xmm11,%xmm11
404e60:  cvtps2pd %xmm3,%xmm5
404e63:  cvtps2pd %xmm11,%xmm13
404e67:  mulpd  %xmm0,%xmm5
404e6b:  mulpd  %xmm0,%xmm13

404e7c:  cvtps2pd %xmm2,%xmm6
404e7f:  cvtps2pd %xmm10,%xmm14
404e83:  addpd  %xmm4,%xmm6
404e87:  addpd  %xmm12,%xmm14
404e8c:  cvtpd2ps %xmm6,%xmm9
404e91:  movhlps %xmm2,%xmm2
404e94:  movhlps %xmm10,%xmm10
404e98:  cvtps2pd %xmm2,%xmm7
404e9b:  cvtps2pd %xmm10,%xmm15
404e9f:  cvtpd2ps %xmm14,%xmm2
404ea4:  addpd  %xmm5,%xmm7
404ea8:  addpd  %xmm13,%xmm15
404ead:  cvtpd2ps %xmm7,%xmm8
404eb2:  cvtpd2ps %xmm15,%xmm15
404eb7:  movlhps %xmm8,%xmm9
404ebb:  movlhps %xmm15,%xmm2

404ecb:  add    $0x8,%r10
404ecf:  cmp    %rbp,%r10
404ed2:  jb     404e3d <saxpy2_+0x14d>
```

```
404e3d:  movaps -0x4(%r9,%r10,4),%xmm3
404e43:  movaps 0xc(%r9,%r10,4),%xmm11

404e59:  movhlps %xmm3,%xmm3
404e5c:  movhlps %xmm11,%xmm11

404e70:  movaps -0x4(%r8,%r10,4),%xmm2
404e76:  movaps 0xc(%r8,%r10,4),%xmm10

404e91:  movhlps %xmm2,%xmm2
404e94:  movhlps %xmm10,%xmm10

404eb7:  movlhps %xmm8,%xmm9
404ebb:  movlhps %xmm15,%xmm2
404ebf:  movaps %xmm9,-0x4(%r9,%r10,4)
404ec5:  movaps %xmm2,0xc(%r9,%r10,4)
404ecb:  add    $0x8,%r10
404ecf:  cmp    %rbp,%r10
404ed2:  jb     404e3d <saxpy2_+0x14d>
```

Differential analysis provides first-order bottlenecks and related ROI (pay-off)

Outline

- Introduction
- MAQAO Toolchain
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules
  - Memory behavior characterization
  - Dynamic bottleneck Analyzer
  - Value profiler
  - PAMDA Methodology
  - Binary Instrumentation Language

Value Profiler

Overview

- Uses tracing (instrumentation)
- Targets:
  - Loop nest instances distribution
  - Loop instances (bound) distribution
  - Function parameters
- Optimization opportunities:
  - Specialization
  - Memoization

Outline

- Introduction
- MAQAO Toolchain
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules
  - Memory behavior characterization
  - Dynamic bottleneck Analyzer
  - Value profiler
  - PAMDA Methodology
  - Binary Instrumentation Language

PAMDA Methodology

Basic blocks

- Top/Down: Decision tree
- Detect hot spots (functions, **loops**)
- Code type characterization:
  - Through dynamic analysis (DECAN)
    - If memory bound: Memory behavior characterization
    - If compute bound: Static analysis
- Value profiling
- Iterative approach: start over again if it is worth

PAMDA Methodology

Overview

1) Basic bandwidth benchmarks (once per architecture)
2) Keep loops up to 80% execution time (sampling)
3) Trace iterations count for each instance
4) Check for short loop trip count
5) Run differential analysis to qualify/quantify bottlenecks (assess if CPU or memory bound…)
6) Run CQA module
7) Investigate CPU or memory bound issues

Outline

- Introduction
- MAQAO Toolchain
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules
  - Memory behavior characterization
  - Dynamic bottleneck Analyzer
  - Value profiler
  - PAMDA Methodology
  - Binary Instrumentation Language

Language concepts/features

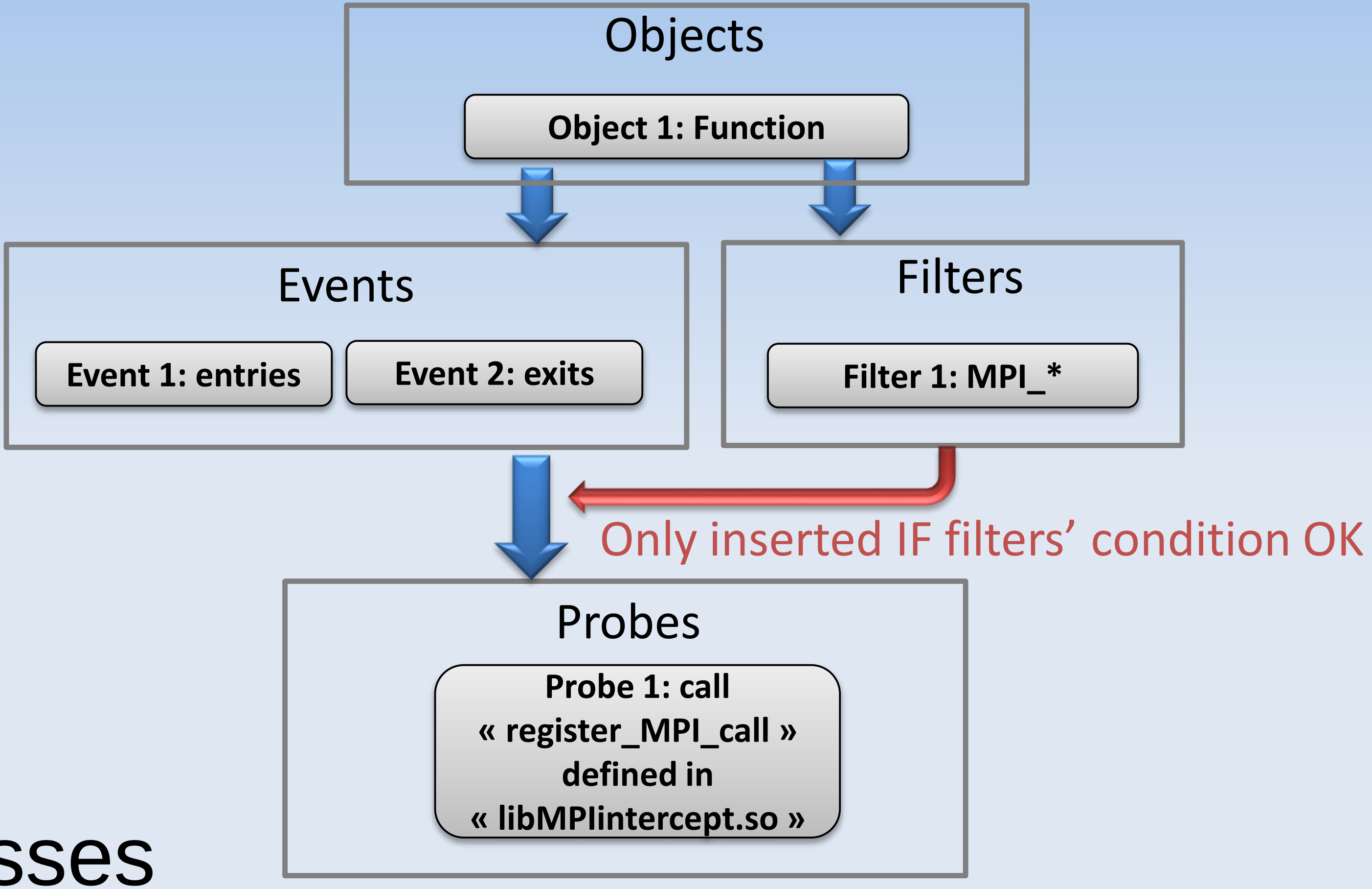
Introduction

- A domain specific language to easily build tools
- Fast prototyping of evaluation tools
  - Easy to use → easy to express → productivity
  - Focus on what (research) and not how (technical)
- Coupling static and dynamic analyses
- Static binary instrumentation
  - Efficient: lowest overhead
  - Robust: ensure the program semantics
  - Accurate: correctly identify program structure

MIL: A language to build program analysis tools through static binary instrumentation.
In *20th Annual International Conference on High Performance Computing*, HiPC'13

Language concepts/features

Overview



Objects

Object 1: Function

Events

Event 1: entries

Event 2: exits

Filters

Filter 1: MPI_*

Only inserted IF filters' condition OK

Probes

Probe 1: call « register_MPI_call » defined in « libMPIintercept.so »

- Objects
- Events
- Probes
- Filters
- Actions
- Variable classes
- Runtime embedded code
- Configuration features (output, properties,etc.)

Language concepts/features

Example

Example 1:
TAU Profiler

- 🟧 **Object**
- 🟦 **Events**
- 🟥 **Probes**
- 🟪 **Configuration**
- 🟩 **Comments**

```
fct_iter = Iterator:new(-1);

this:setRunDir("output_path/");
mb = this:addBinaryMain("./bt.S");
mb:setOutputSuffix("_i");
--Program entry probe
e_exit = mb:newEvent("at_exit");
p_exit = e_exit:newProbeExt("tau_cleanup","libTau.so");
--Instrumentation at function level
fct = mb:addFunction();
--Probe at function entries
e_entries = fct:newEvent("entries");
p_entries = e_entries:newProbeExt("traceEntry","libTau.so");
p_entries:addParamIterCurr(fct_iter);
--Special event to fill Binary:at_entry from function level
e_ape = p_entries:newEvent("at_program_entry");
p_ape = e_ape:newProbeExt("trace_register_func","libTau.so");
p_ape:addParamIterNext(fct_iter);
--Probe at function exits
e_exits = fct:newEvent("exits");
p_exits = e_exits:newProbeExt("traceExit","libTau.so");
p_exits:addParamIterCurr(fct_iter);
```

Thanks for your attention !

Questions ?