MAQAO
Performance Analysis and Optimization Tool

Cédric VALENSI, Emmanuel OSERET, Salah IBN AMAR
{cedric.valensi, emmanuel.oseret, mohammed-salah.ibnamar}@uvsq.fr
Performance Evaluation Team, University of Versailles S-Q-Y
http://www.maqao.org
VI-HPS 29th Reims – France – 15-19 September 2018

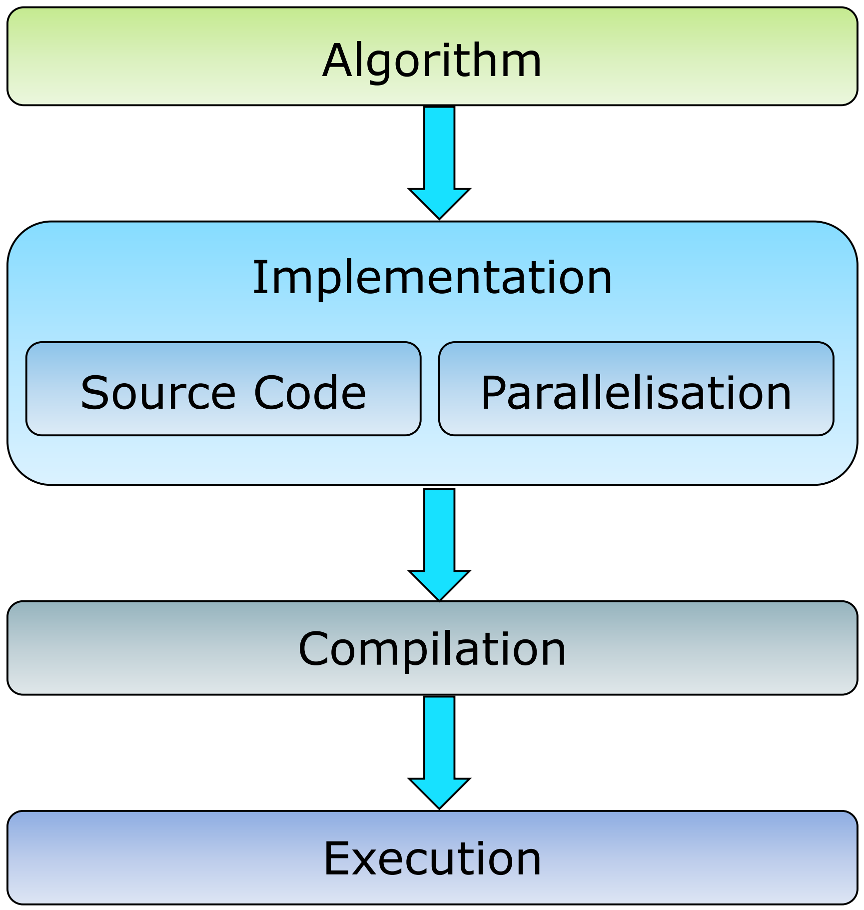
Performance analysis and optimisation

How much can I optimise my application?
- Can it actually be done?
- What would the effort/gain ratio be?

Where can I gain time?
- Where is my application wasting time?

Why is the application spending time there?
- Algorithm, implementation or hardware?
- Data access or computation?

How can I improve the situation?
- In which step(s) of the design process?
- What additional information do I need?



Motivating example

Code of a loop representing ~10% walltime

6) Variable number of iterations

2) Non-unit stride accesses

1) High number of statements

```
do j = ni + nvalue1, nato

    nj1 = ndim3d*j + nc ; nj2 = nj1 + nvalue1 ; nj3 = nj2 + nvalue1
    u1 = x11 – x(nj1) ; u2 = x12 – x(nj2) ; u3 = x13 – x(nj3)
    rtest2 = u1*u1 + u2*u2 + u3*u3 ; cnij = eci*qEold(j)
    rij = demi*(rvwi + rvwalc1(j))
    drtest2 = cnij/(rtest2 + rij) ; drtest = sqrt(drtest2)
    Eq = qq1*qq(j)*drtest
    ntj = nti + ntype(j)
    Ed = ceps(ntj)*drtest2*drtest2*drtest2
    Eqc = Eqc + Eq ; Ephob = Ephob + Ed
    gE = (c6*Ed + Eq)*drtest2 ; virt = virt + gE*rtest2
    u1g = u1*gE ; u2g = u2*gE ; u3g = u3*gE
    g1c = g1c –u1g ; g2c = g2c – u2g ; g3c = g3c –u3g
    gr(nj1, thread_num) = gr(nj1, thread_num) + u1g
    gr(nj2, thread_num) = gr(nj2, thread_num) + u2g
    gr(nj3, thread_num) = gr(nj3, thread_num) + u3g

end do
```

4) DIV/SQRT

3) Indirect accesses

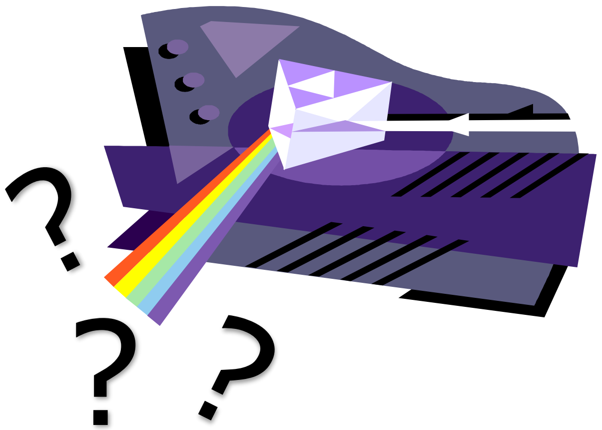
5) Reductions

2) Non-unit stride accesses

Source code and associated issues:

1) High number of statements
2) Non-unit stride accesses
3) Indirect accesses
4) DIV/SQRT
5) Reductions
6) Variable number of iterations

A multifaceted problem



Pinpointing the performance bottlenecks

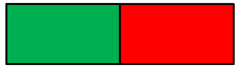
Identifying the dominant issues

- Algorithms, implementation, parallelisation, …

Making the **best use** of the machine features

- Complex multicore and manycore CPUs
- Complex memory hierarchy

Finding the **most rewarding** issues to be fixed

- **40%** total time, expected **10%** speedup
  - ➔ TOTAL IMPACT: **4%** speedup 
- **20%** total time, expected **50%** speedup
  - ➔ TOTAL IMPACT: **10%** speedup 

=> Need for dedicated and complementary tools

MAQAO: Modular Assembly Quality Analyzer and Optimizer

Objectives:

- Characterizing performance of HPC applications
- Focusing on performance at the **core level**
- **Guiding users** through optimization process
- Estimating return of investment (**R.O.I.**)

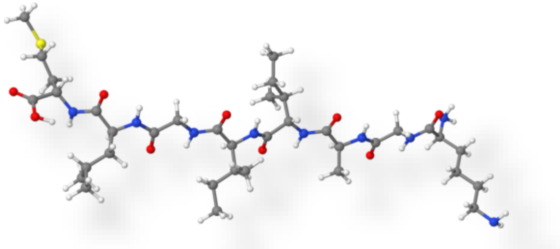
Characteristics:

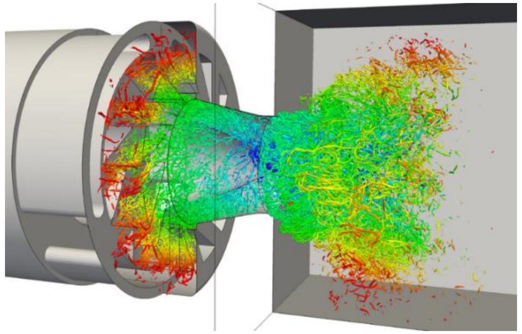
- **Modular tool** offering complementary views
- Support for **Intel x86-64** and **Xeon Phi**
  - ARM under development
- LGPL3 Open Source software
- Developed at UVSQ since 2004
- Binary release available as **static executable**

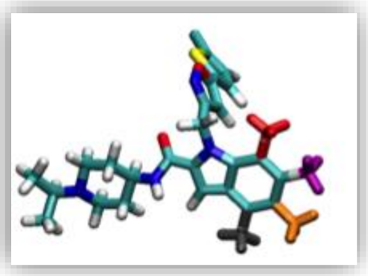
Success stories

MAQAO was used for optimizing industrial and academic HPC applications:

- QMC=CHEM (IRSAMC)
  - Quantum chemistry
  - Speedup: **> 3x**
    - Moved invocation of function with identical parameters out of loop body
- Yales2 (CORIA)
  - Computational fluid dynamics
  - Speedup: **up to 2.8x**
    - Removed double structure indirections
- Polaris (CEA)
  - Molecular dynamics
  - Speedup: **1.5x – 1.7x**
    - Enforced loop vectorisation through compiler directives
- AVBP (CERFACS)
  - Computational fluid dynamics
  - Speedup: **1.08x – 1.17x**
    - Replaced division with multiplication by reciprocal
    - Complete unrolling of loops with small number of iterations







Partnerships

MAQAO was funded by UVSQ, Intel and CEA (French department of energy) through Exascale Computing Research (ECR) and the French Ministry of Industry through various FUI/ITEA projects (H4H, COLOC, PerfCloud, ELCI, etc...)

Provides core technology to be integrated with other tools:
- TAU performance tools with MADRAS patcher through MIL (MAQAO Instrumentation Language)
- ATOS bullxprof with MADRAS through MIL
- Intel Advisor
- INRIA Bordeaux HWLOC

PeXL ISV also contributes to MAQAO:
- Commercial performance optimization expertise
- Training and software development
- www.pexl.eu

MAQAO team and collaborators

- Prof. William Jalby
- *Prof. Denis Barthou*
- Prof. David J. Kuck
- Andrés S. Charif-Rubial, Ph D
- *Jean-Thomas Acquaviva, Ph D*
- *Stéphane Zuckerman, Ph D*
- *Julien Jaeger, Ph D*
- *Souad Koliaï, Ph D*
- Cédric Valensi, Ph D
- Eric Petit, Ph D
- *Zakaria Bendifallah, Ph D*
- Emmanuel Oseret, Ph D
- Pablo de Oliveira, Ph D
- *Tipp Moseley, Ph D*
- David C. Wong, Ph D
- *Jean-Christophe Beyler, Ph D*
- Mathieu Tribalat
- Hugo Bolloré
- *Jean-Baptiste Le Reste*
- *Sylvain Henry, Ph D*
- Salah Ibn Amar
- Youenn Lebras
- Othman Bouizi, Ph D
- *José Noudohouenou, Ph D*
- Aleksandre Vardoshvili
- Romain Pillot

Analysis at binary level

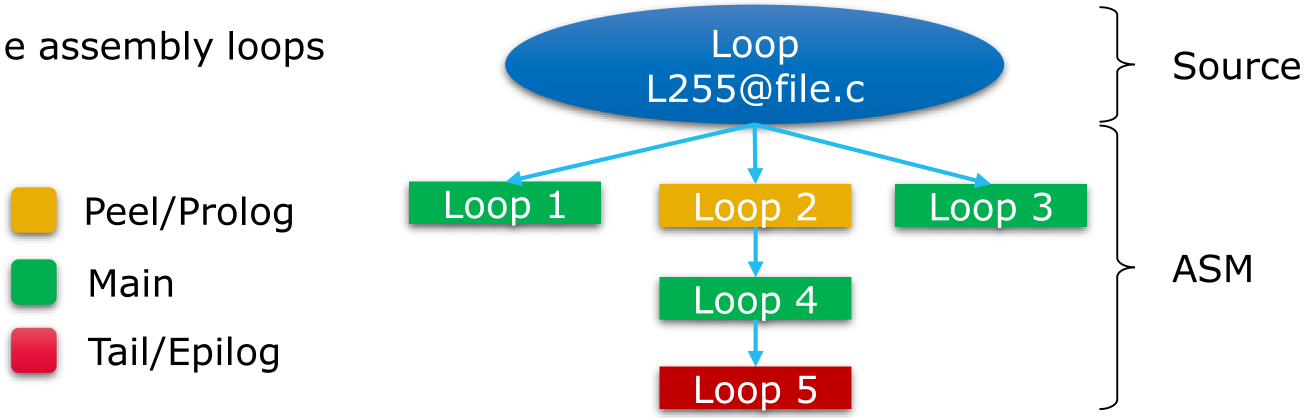
Advantages of binary analysis:

- Compiler optimizations increase the distance between the executed code and the source
- Source code instrumentation may prevent the compiler from applying some transformations

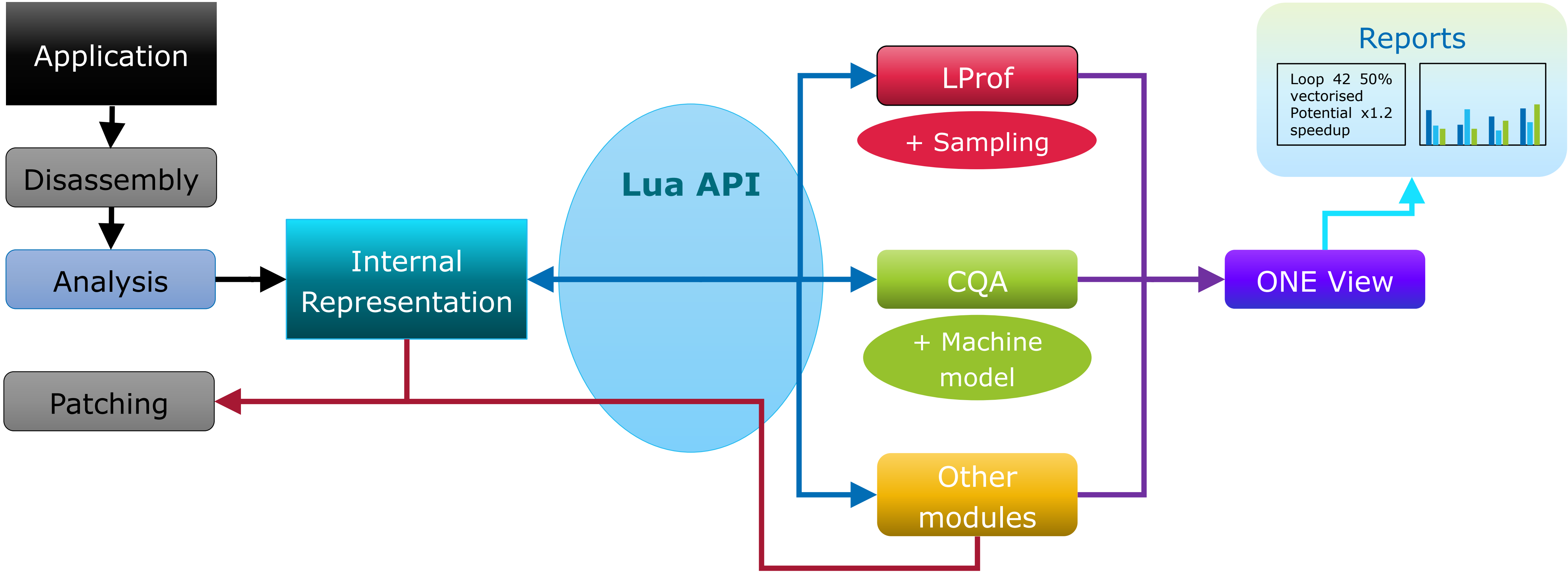
We want to evaluate the "real" executed code: **What You Analyse Is What You Run**

Main steps:

- Reconstruct the program structure
- Relate the analyses to source code
  - A single source loop can be compiled as multiple assembly loops
  - Affecting unique identifiers to loops

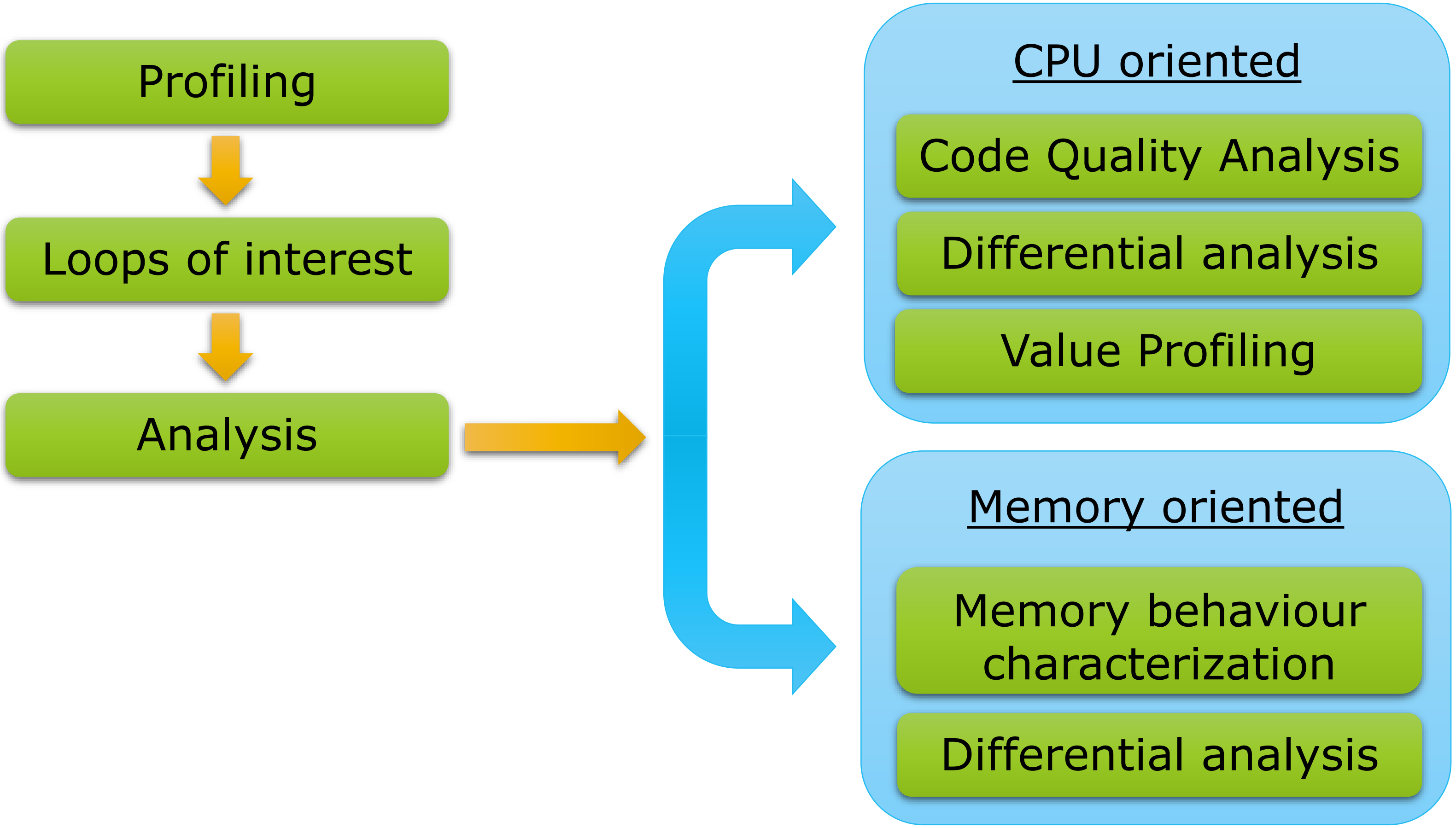


MAQAO Main structure



MAQAO Methodology

Decision tree



MAQAO LProf: Lightweight Profiler

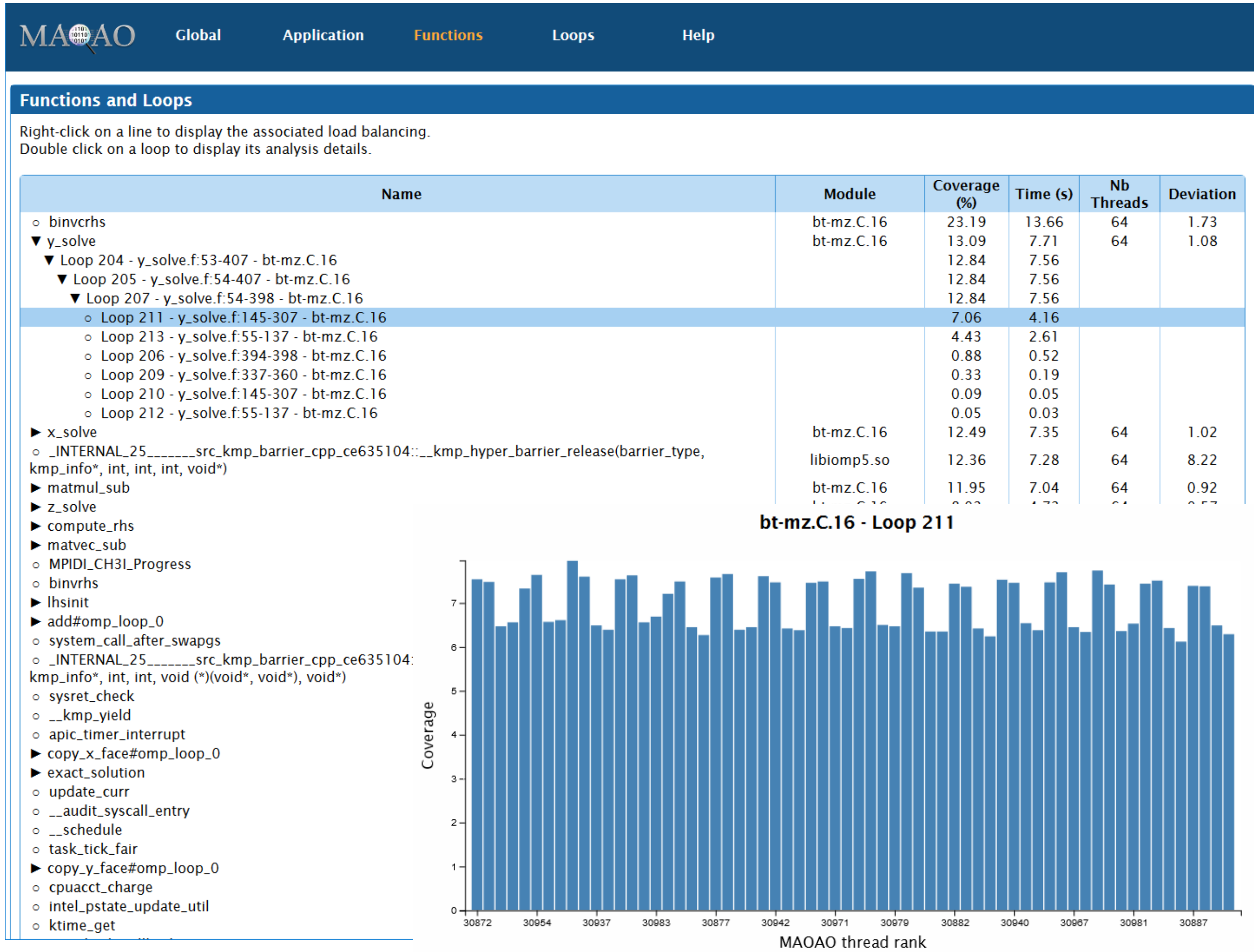
Goal: Lightweight localization of application hotspots

Features:

- **Sampling** based
- Access to hardware counters for additional information
- Results at function and loop granularity

Strengths:

- **Non intrusive**: No recompilation necessary
- **Low overhead**
- Agnostic with regard to parallel runtime



MAQAO CQA: Code Quality Analyzer

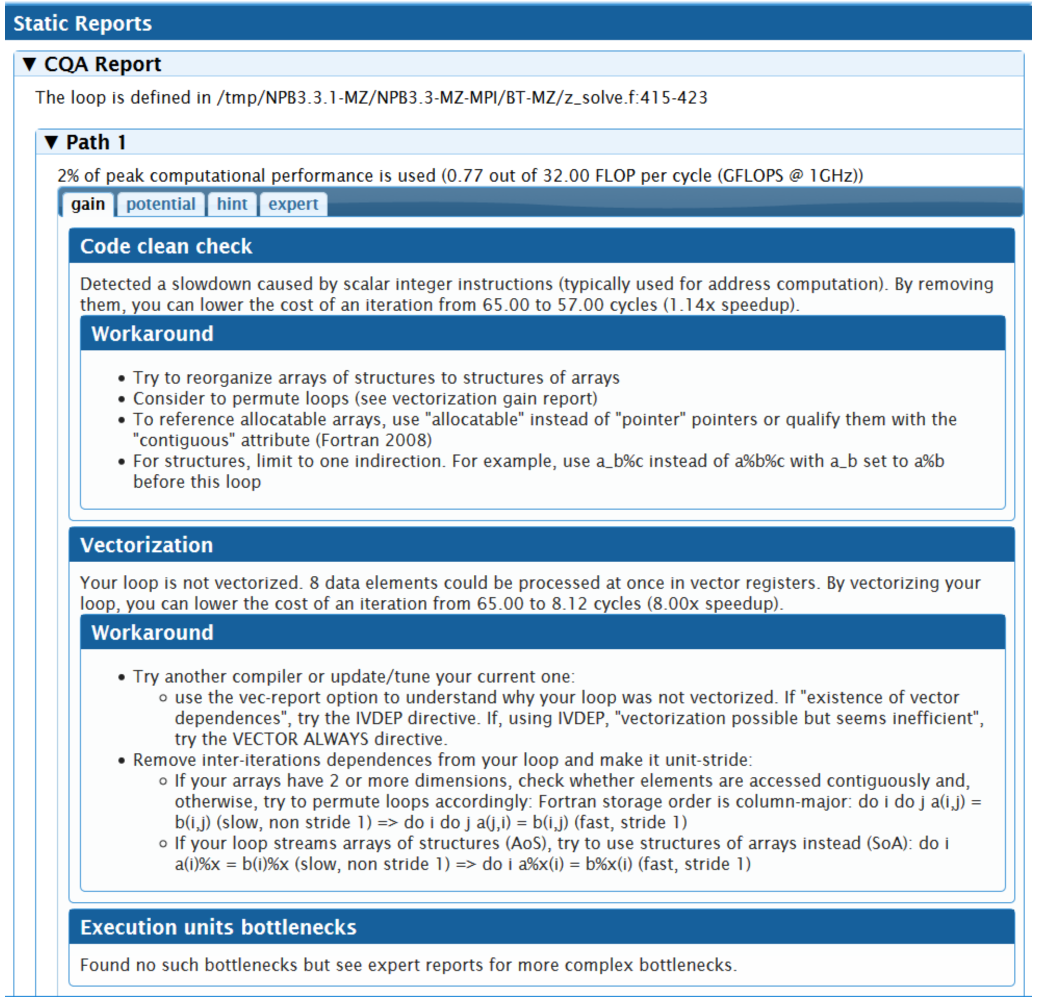
Goal: **Assist developers** in improving code performance

Features:

- Evaluates the **quality** of the compiler generated code
- Returns **hints and workarounds** to improve quality
- Focuses on **loops**
  - In HPC most of the time is spent in loops
- Targets **compute-bound** codes

Static analysis:

- Requires **no execution** of the application
- Allows **cross-analysis**



Static Reports

▼ CQA Report

The loop is defined in /tmp/NPB3.3.1-MZ/NPB3.3-MZ-MPI/BT-MZ/z_solve.f:415-423

▼ Path 1

2% of peak computational performance is used (0.77 out of 32.00 FLOP per cycle (GFLOPS @ 1GHz))

gain | potential | hint | expert

Code clean check

Detected a slowdown caused by scalar integer instructions (typically used for address computation). By removing them, you can lower the cost of an iteration from 65.00 to 57.00 cycles (1.14x speedup).

Workaround

- Try to reorganize arrays of structures to structures of arrays
- Consider to permute loops (see vectorization gain report)
- To reference allocatable arrays, use "allocatable" instead of "pointer" pointers or qualify them with the "contiguous" attribute (Fortran 2008)
- For structures, limit to one indirection. For example, use a_b%c instead of a%b%c with a_b set to a%b before this loop

Vectorization

Your loop is not vectorized. 8 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 65.00 to 8.12 cycles (8.00x speedup).

Workaround

- Try another compiler or update/tune your current one:
  - 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:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly: Fortran storage order is column-major: do i do j a(i,j) = b(i,j) (slow, non stride 1) => do i do j a(j,i) = b(i,j) (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA): do i a(i)%x = b(i)%x (slow, non stride 1) => do i a%x(i) = b%x(i) (fast, stride 1)

Execution units bottlenecks

Found no such bottlenecks but see expert reports for more complex bottlenecks.

MAQAO CQA: Main Concepts

Most of the time, applications only exploit at best 5 to 10% of the peak performance.

Main elements of analysis:
- Peak performance
- Execution pipeline
- Resources/Functional units

Key performance levers for core level efficiency:
- Vectorizing
- Avoiding high latency instructions if possible
- Having the compiler generate an efficient code
- Reorganizing memory layout

Same instruction – Same cost

Process up to 8X data

MAQAO CQA: Compiler and programmer hints

Compiler can be driven using flags and pragmas:

- Ensuring full use of architecture capabilities (e.g. using flag -xHost on AVX capable machines)
- Forcing optimization (unrolling, vectorization, alignment, …)
- Bypassing conservative behaviour when possible (e.g. 1/X precision)

Implementation changes:

- Improve data access
  - Loop interchange
  - Changing loop strides
  - Reshaping arrays of structures
- Avoid instructions with high latency

MAQAO ONE View: Performance View Aggregator

Automating the whole analysis process

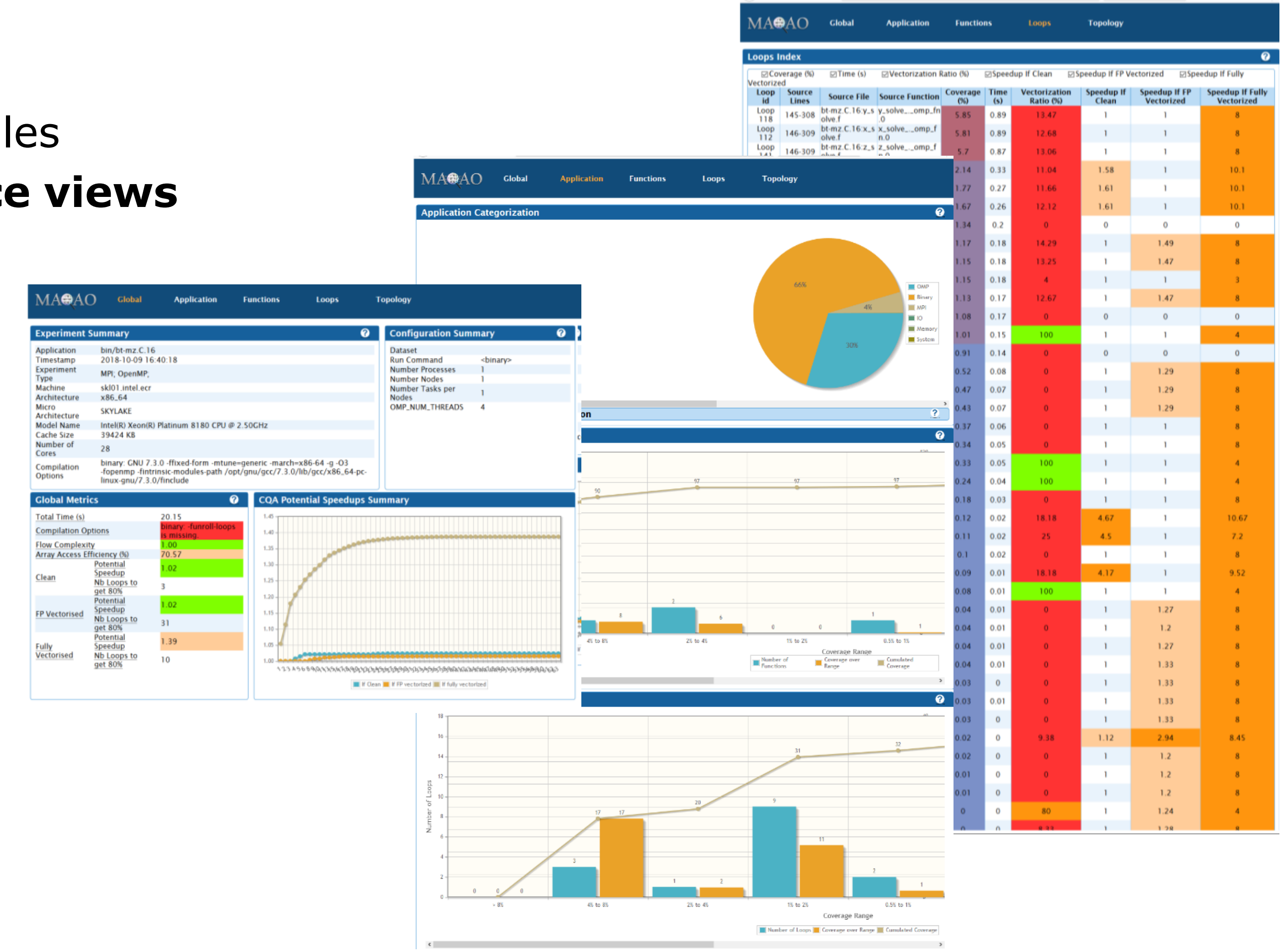
- Invocation of the required MAQAO modules
- Generation of **aggregated performance views** available as HTML files

Main steps:

- Invokes LProf to **identify hotspots**
- Invokes CQA on **loop hotspots**

Available results:

- **Speedup** predictions
- Global code **quality** metrics
- **Hints** for improving performance



Analysing an application with MAQAO

Execute ONE View

- Provide all parameters necessary for executing the application
  - Parameters can be passed on the command line or into a configuration file

```
$ maqao oneview --create-report=one --binary=bt-mz.C.16 --mpi_command="mpirun -n 16"
```

- Analyses can be tweaked if necessary
  - Report one corresponds to profiling and code quality analysis
- ONE View can reuse an existing experiment directory to perform further analyses
- Results available in HTML by default
  - XLS files or console output available

MAQAO modules can be invoked separately for advanced analyses

```
$ maqao lprof xp=exp_dir --mpi-command="mpirun -n 16" -- ./bt-mz.C.16 # Profiling
$ maqao lprof xp=exp_dir -df                                   # Displays results
```

```
$ maqao cqa loop=42 bt-mz.C.16
```

Help and tutorials available on the MAQAO website: www.maqao.org/documentation.html

Global summary

Experiment summary

- Characteristics of the machine where the experiment took place

Global metrics

- General quality metrics derived from MAQAO analyses
- Global speedup predictions

CQA potential speedups

- Speedup prediction depending on the number of vectorised loops
- Ordered speedups to identify the loops to optimise in priority

Application Characteristics

Application categorisation
- Time spent in different regions of code

Function based profile
- Functions by coverage ranges

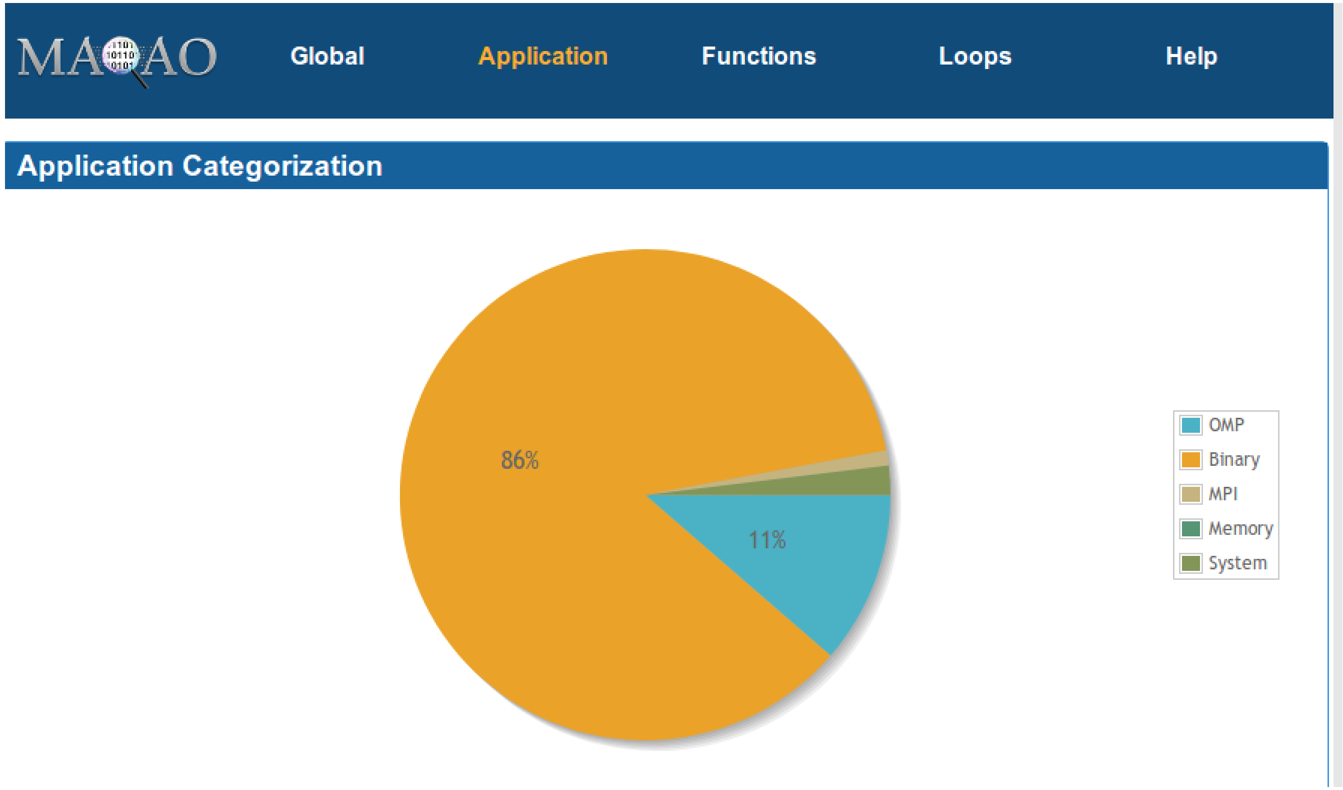
Loop based profile
- Loops by coverage ranges

Detailed loop based profile
- Loop types by coverage ranges

Application Characteristics: Time Categorisation

Identifying at a glance where time is spent

- Application
  - Main executable
- Parallelization
  - Threads
  - OpenMP
  - MPI
- System libraries
  - I/O operations
  - String operations
  - Memory management functions
- External libraries
  - Specialised libraries such as libm / libmkl
  - Application code in external libraries

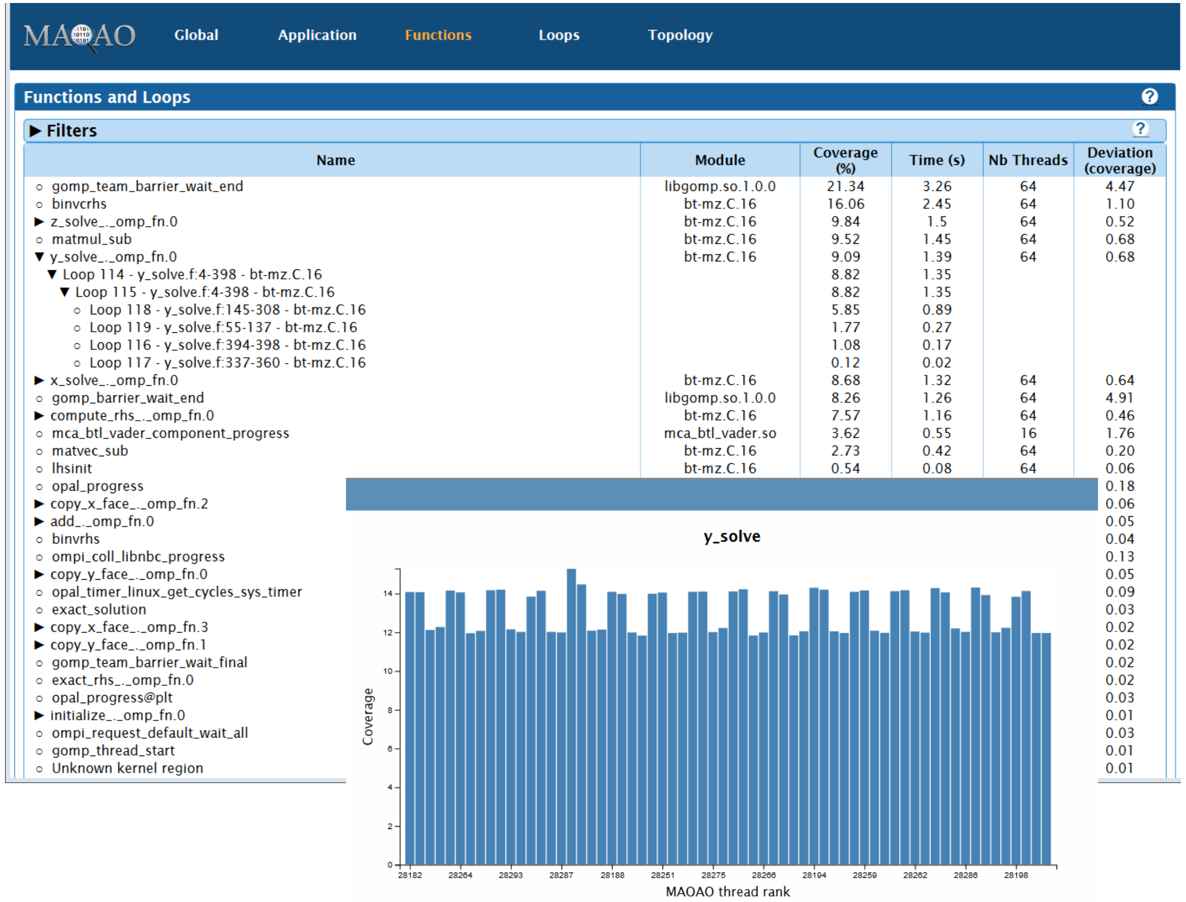


Functions Profiling

Identifying hotspots

- Exclusive coverage
- Load balancing across threads
- Loops nests by functions

- ▼ matmul_sub
  - ○ Loop 230 - solve_subs.f:71-175 - bt-mz.C.16
  - ○ Loop 231 - solve_subs.f:71-175 - bt-mz.C.16 — Single
- ▼ z_solve
  - ▼ Loop 232 - z_solve.f:53-423 - bt-mz.C.16 — Outermost
    - ▼ Loop 233 - z_solve.f:54-423 - bt-mz.C.16 — Inbetween
      - ▼ Loop 236 - z_solve.f:54-423 - bt-mz.C.16 — Inbetween
        - ○ Loop 239 - z_solve.f:146-308 - bt-mz.C.16 — Innermost
        - ○ Loop 235 - z_solve.f:55-137 - bt-mz.C.16
        - ○ Loop 234 - z_solve.f:415-423 - bt-mz.C.16



Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)
○ gomp_team_barrier_wait_end	libgomp.so.1.0.0	21.34	3.26	64	4.47
○ binvcrhs	bt-mz.C.16	16.06	2.45	64	1.10
▶ z_solve_._omp_fn.0	bt-mz.C.16	9.84	1.5	64	0.52
○ matmul_sub	bt-mz.C.16	9.52	1.45	64	0.68
▼ y_solve_._omp_fn.0	bt-mz.C.16	9.09	1.39	64	0.68
▼ Loop 114 - y_solve.f:4-398 - bt-mz.C.16		8.82	1.35		
▼ Loop 115 - y_solve.f:4-398 - bt-mz.C.16		8.82	1.35		
○ Loop 118 - y_solve.f:145-308 - bt-mz.C.16		5.85	0.89		
○ Loop 119 - y_solve.f:55-137 - bt-mz.C.16		1.77	0.27		
○ Loop 116 - y_solve.f:394-398 - bt-mz.C.16		1.08	0.17		
○ Loop 117 - y_solve.f:337-360 - bt-mz.C.16		0.12	0.02		
▶ x_solve_._omp_fn.0	bt-mz.C.16	8.68	1.32	64	0.64
○ gomp_barrier_wait_end	libgomp.so.1.0.0	8.26	1.26	64	4.91
▶ compute_rhs_._omp_fn.0	bt-mz.C.16	7.57	1.16	64	0.46
○ mca_btl_vader_component_progress	mca_btl_vader.so	3.62	0.55	16	1.76
○ matvec_sub	bt-mz.C.16	2.73	0.42	64	0.20
○ lhsinit	bt-mz.C.16	0.54	0.08	64	0.06
○ opal_progress					0.18
▶ copy_x_face_._omp_fn.2					0.06
▶ add_._omp_fn.0					0.05
○ binvrhs					0.04
○ ompi_coll_libnbc_progress					0.13
▶ copy_y_face_._omp_fn.0					0.05
○ opal_timer_linux_get_cycles_sys_timer					0.09
○ exact_solution					0.03
▶ copy_x_face_._omp_fn.3					0.02
▶ copy_y_face_._omp_fn.1					0.02
○ gomp_team_barrier_wait_final					0.02
○ exact_rhs_._omp_fn.0					0.02
○ opal_progress@plt					0.03
▶ initialize_._omp_fn.0					0.01
○ ompi_request_default_wait_all					0.03
○ gomp_thread_start					0.01
○ Unknown kernel region					0.01

Loops Profiling Summary

Identifying loop hotspots

- Vectorisation information
- Potential speedups

MAQAO Global Application Functions Loops Help

Loops Index

Double click on a loop to display its analysis details.
Check / uncheck metrics above the table to display / hide them.

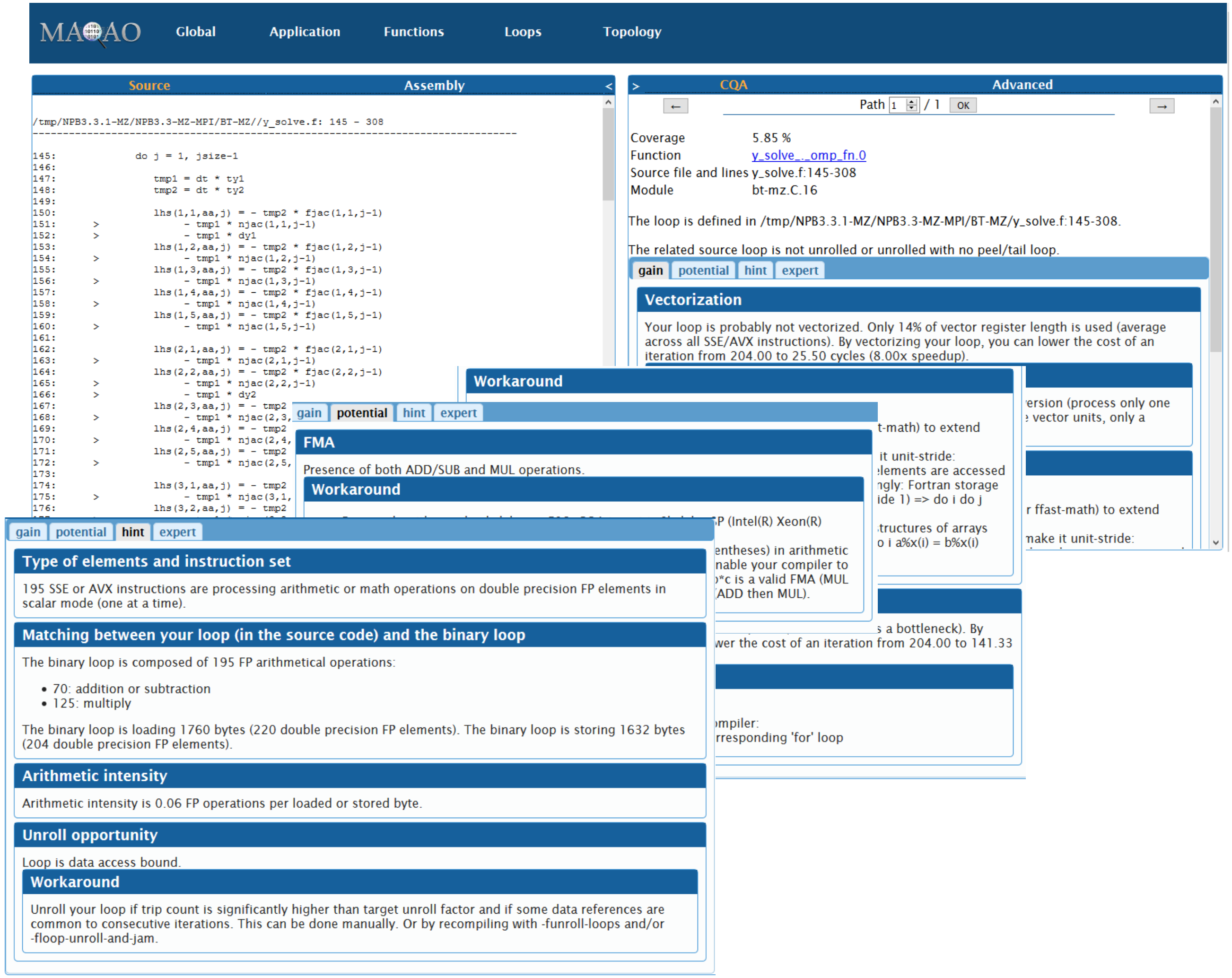
☑Coverage (%) ☑Vectorization Ratio (%) ☑Speedup If Clean ☑Speedup If FP Vectorized ☑Speedup If Fully Vectorized

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 211	145-307	bt-mz.C.16:y_solve.f	y_solve	7.06	45.13	1	1.22	5.52
Loop 201	146-308	bt-mz.C.16:x_solve.f	x_solve	7.06	45.13	1	1.22	5.52
Loop 230	71-175	bt-mz.C.16:solve_subs.f	matmul_sub	5.57	100	1.02	1.9	4
Loop 213	55-137	bt-mz.C.16:y_solve.f	y_solve	4.43	47	1.05	1.16	5.93
Loop 203	57-139	bt-mz.C.16:x_solve.f	x_solve	3.93	48.36	1.01	1.09	5.83
Loop 239	146-308	bt-mz.C.16:z_solve.f	z_solve	3.06	8.97	1.07	1.8	8
Loop 235	55-137	bt-mz.C.16:z_solve.f	z_solve	2.81	22.08	1.03	1.62	7.49
Loop 234	415-423	bt-mz.C.16:z_solve.f	z_solve	1.54	0	1.14	1.67	8
Loop 122	304-349	bt-mz.C.16:rhs.f	compute_rhs	1.32	71.26	1.33	1.92	5.36
Loop 148	194-238	bt-mz.C.16:rhs.f	compute_rhs	1.25	71.59	1.32	1.93	5.38
Loop 162	83-132	bt-mz.C.16:rhs.f	compute_rhs	1.23	71.59	1.24	1.87	5.26
Loop 231	71-175	bt-mz.C.16:solve_subs.f	matmul_sub	1.11	10.59	1	2.29	8
Loop 227	23-27	bt-mz.C.16:solve_subs.f	matvec_sub	0.97	100	1	1.95	4
Loop 206	394-398	bt-mz.C.16:y_solve.f	y_solve	0.88	0	1.04	1.73	8
Loop 196	395-399	bt-mz.C.16:x_solve.f	x_solve	0.84	0	1.04	2.02	8
Loop 229	23-27	bt-mz.C.16:solve_subs.f	matvec_sub	0.62	100	1	1.95	4
Loop 170	40-50	bt-mz.C.16:rhs.f	compute_rhs	0.4	73.33	1	1.82	4
Loop 105	388-391	bt-mz.C.16:rhs.f	compute_rhs	0.35	100	1.12	1.83	4
Loop 238	313-314	bt-mz.C.16:z_solve.f	z_solve	0.35	0	1	1	8

Loop Analysis Reports

High level reports

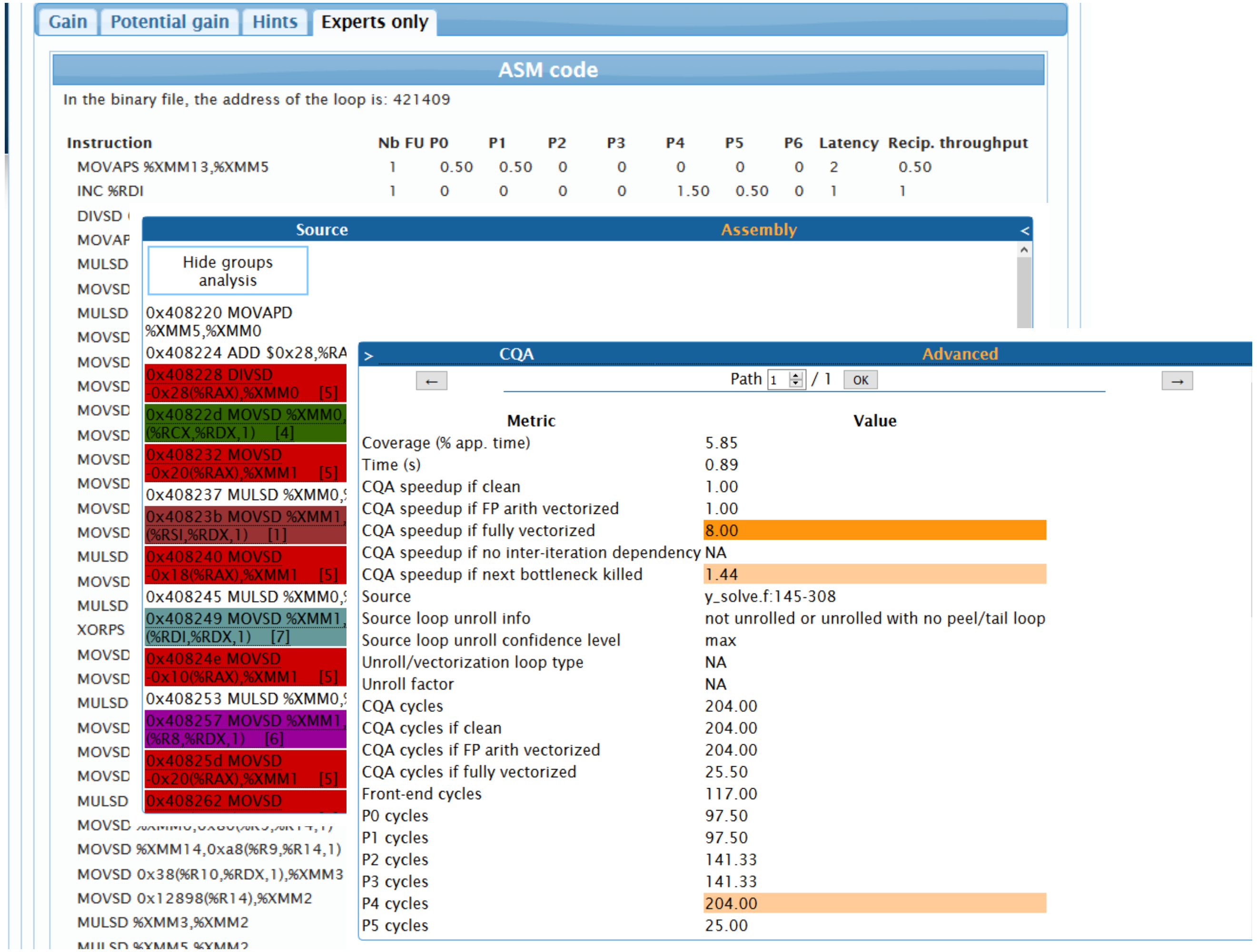
- Reference to the source code
- Bottleneck description
- Hints for improving performance
- Reports categorized by probability that applying hints will yield predicted gain
  - Gain: Good probability
  - Potential gain: Average probability
  - Hints: Lower probability



Loop Analysis Reports – Expert View

Low level reports for performance experts

- Assembly-level
- Instructions cycles costs
- Instructions dispatch predictions
- Memory access analysis



Application to Motivating Example

Issues identified by CQA

6) Variable number of iterations

2) Non-unit stride accesses

```
do j = ni + nvalue1, nato

   nj1 = ndim3d*j + nc ; nj2 = nj1 + nvalue1 ; nj3 = nj2 + nvalue1
   u1 = x11 – x(nj1) ; u2 = x12 – x(nj2) ; u3 = x13 – x(nj3)
   rtest2 = u1*u1 + u2*u2 + u3*u3 ; cnij = eci*qEold(j)
   rij = demi*(rvwi + rvwalc1(j))
   drtest2 = cnij/(rtest2 + rij) ; drtest = sqrt(drtest2)
   Eq = qq1*qq(j)*drtest
   ntj = nti + ntype(j)
   Ed = ceps(ntj)*drtest2*drtest2*drtest2
   Eqc = Eqc + Eq ; Ephob = Ephob + Ed
   gE = (c6*Ed + Eq)*drtest2 ; virt = virt + gE*rtest2
   u1g = u1*gE ; u2g = u2*gE ; u3g = u3*gE
   g1c = g1c –u1g ; g2c = g2c – u2g ; g3c = g3c –u3g
   gr(nj1, thread_num) = gr(nj1, thread_num) + u1g
   gr(nj2, thread_num) = gr(nj2, thread_num) + u2g
   gr(nj3, thread_num) = gr(nj3, thread_num) + u3g

end do
```

1) High number of statements

7) Vector vs scalar

4) DIV/SQRT

3) Indirect accesses

5) Reductions

2) Non-unit stride accesses

CQA can detect and provide hints to resolve most of the identified issues:

1) **High number of statements**
2) **Non-unit stride accesses**
3) **Indirect accesses**
4) **DIV/SQRT**
5) Reductions
6) Variable number of iterations
7) **Vector vs scalar**

Application to Motivating Example

Gain | Potential gain | Hints | Experts only

Vectorization

Your loop is partially vectorized.
Only 28% of vector register length is used (average across all SSE/AVX instructions).
By fully vectorizing your loop, you can lower the cost of an iteration from 57.00 to 21.50 cycles (2.65x speedup).
51% of SSE/AVX instructions are used in vector version (process two or more data elements in vector registers):

- *24% of SSE/AVX loads are used in vector version.*
- *0% of SSE/AVX stores are used in vector version.*

Since your execution units are vector units, only a fully vectorized loop can use their full power.

Proposed solution(s):

- Try another compiler or update/tune your current one:
  - 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:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly:
    Fortran storage order is column-major: do i do j a(i,j) = b(i,j) (slow, non stride 1) => do i do j a(j,i) = b(i,j) (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA):
    do i a(i)%x = b(i)%x (slow, non stride 1) => do i a%x(i) = b%x(i) (fast, stride 1)

Execution units bottlenecks

Performance is limited by:

- execution of divide and square root operations (the divide/square root unit is a bottleneck)
- execution of INT/FP operations on vector registers (the VPU is a bottleneck)

By removing all these bottlenecks, you can lower the cost of an iteration from 57.00 to 48.00 cycles (1.19x speedup).

Proposed solution(s):

- Reduce the number of division or square root instructions.
  If denominator is constant over iterations, use reciprocal (replace x/y with x*(1/y)). Check precision impact. This will be done by your compiler with no-prec-div or Ofast.
  Check whether you really need double precision. If not, switch to single precision to speedup execution.
- Reduce arithmetical operations on array elements

Gain | Potential gain | Hints | Experts only

FMA

Detected 48 FMA (fused multiply-add) operations.
Presence of both ADD/SUB and MUL operations.

Proposed solution(s):

Try to change order in which elements are evaluated (using parentheses) in arithmetic expressions containing both ADD/SUB and MUL operations to enable your compiler to generate FMA instructions wherever possible.
For instance a + b*c is a valid FMA (MUL th
However (a+b)* c cannot be translated into

Gain | Potential gain | Hints | Experts only

Slow data structures access

Detected data structures (typically arrays) that cannot be efficiently read/written

- *Constant non-unit stride: 1 occurrence(s)*
- *Irregular (variable stride) or indirect: 1 occurrence(s)*

1) High number of statements
2) Non-unit stride accesses
3) Indirect accesses
4) DIV/SQRT
5) Reductions
6) Variable number of iterations
7) Vector vs scalar

Thank you for your attention !

Questions ?