MAQAO
Performance Analysis and Optimization Tool

Cédric VALENSI, Emmanuel OSERET
cedric.valensi@uvsq.fr, emmanuel.oseret@uvsq.fr
Performance Evaluation Team, University of Versailles S-Q-Y
http://www.maqao.org

VI-HPS 23rd Livermore/CA – USA – 27-29 July 2016

Introduction

Objectives:

- Performance characterization of HPC applications
- Focus optimization efforts
- Estimation of R.O.I.

Main functionalities:

- Profiling and hardware counters collection
- Code quality analysis

Characteristics:

- Modular tool
- Support for Intel x86-64 and Xeon Phi
- LGPL3 Open Source software
- Developed at UVSQ since 2004

Introduction
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...)

Optimizes industrial and academic HPC applications:

- Yales2, AVBP, Polaris, QMC=CHEM, …

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 AmplifierXE

Introduction
Some MAQAO Collaborators

- Prof. William Jalby
- Prof. Denis Barthou
- 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
- 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 Noudohouennou, Ph D
- …

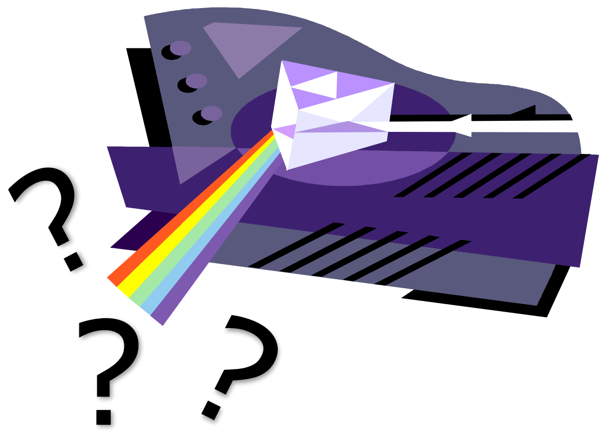
Introduction
Performance analysis (1/2)

Characterizing application performance:

- Profiling application
- Pinpointing the performance bottlenecks
  - Complex multicore and manycore CPUs
  - Complex memory hierarchy
- Making best use of the machine features

Facing a multifaceted problem:

- How to determine the dominant issues?
  - Algorithms choice
  - Implementation
  - Parallelization
  - …
- Maximizing the number of views



=> Need for dedicated and complementary tools

Introduction
Performance analysis (2/2)

Motivating example: loop ~10% walltime

```
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
```

Annotations: Variable number of iterations; Non-unit stride accesses; DIV/SQRT; Indirect accesses; Reductions; Non-unit stride accesses; High number of statements vector versus calar

Source code and associated issues:

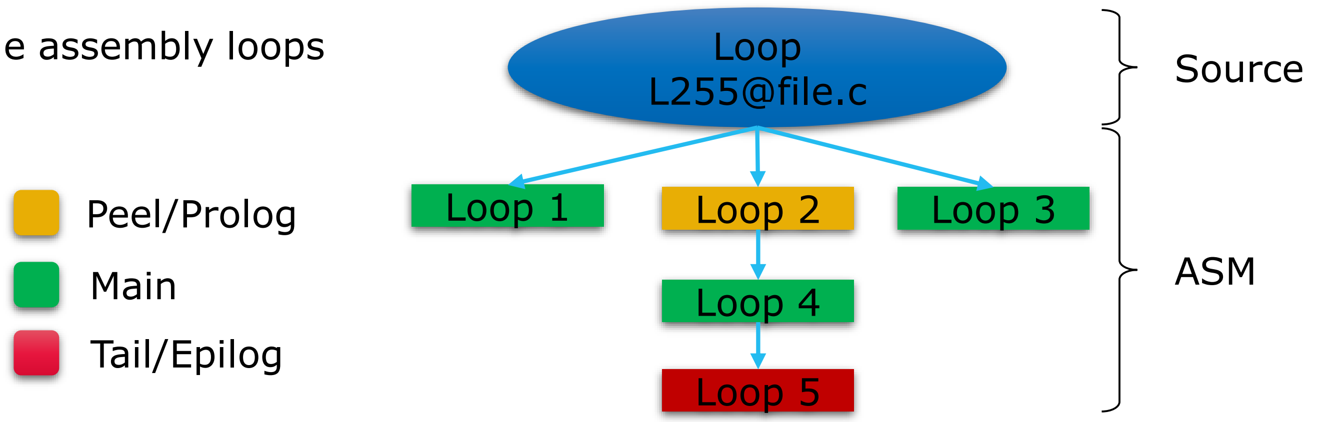
1) High number of statements
2) Non-unit stride accesses
3) Indirect accesses
4) DIV/SQRT
5) Reductions
6) Vector vs Scalar

Introduction
MAQAO: Analysis at binary level

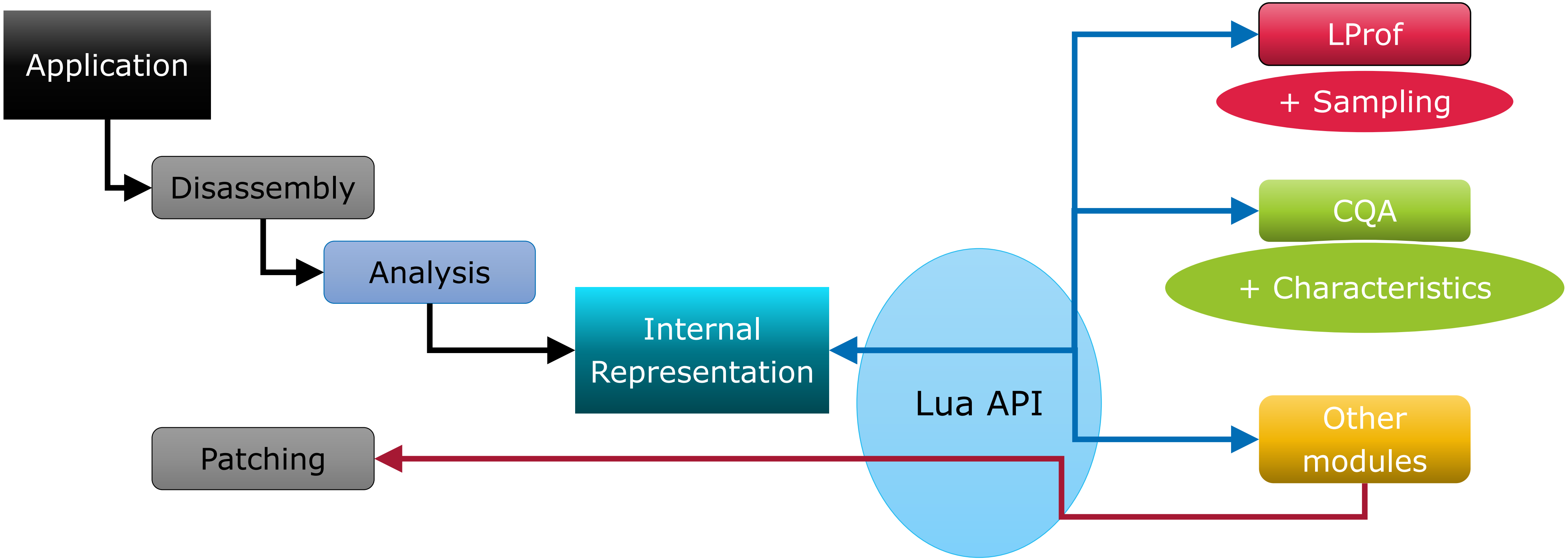
Advantages of binary analysis:
- Compiler optimizations increase the distance between the executed code and the source
- Source code instrumentation may prevent the compiler from applying some transformations

We want to evaluate the "real" executed code: What You Analyze 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

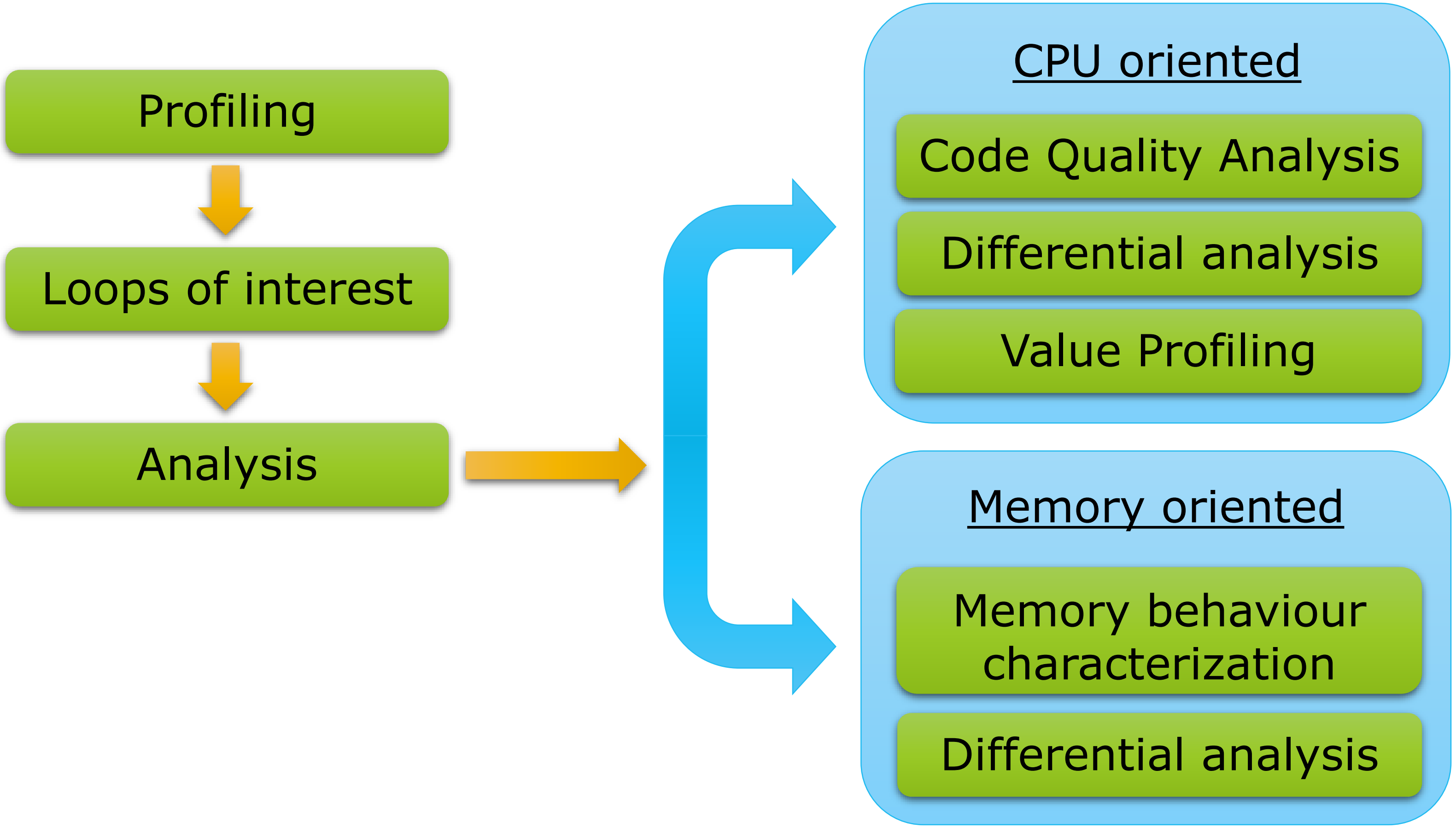


Introduction
MAQAO Main Structure

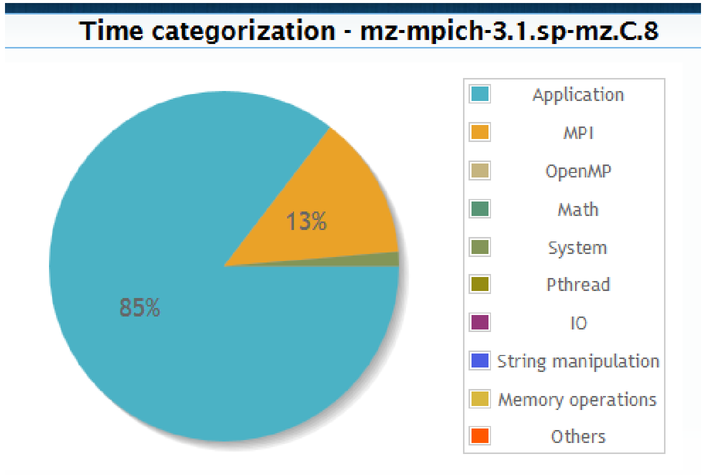


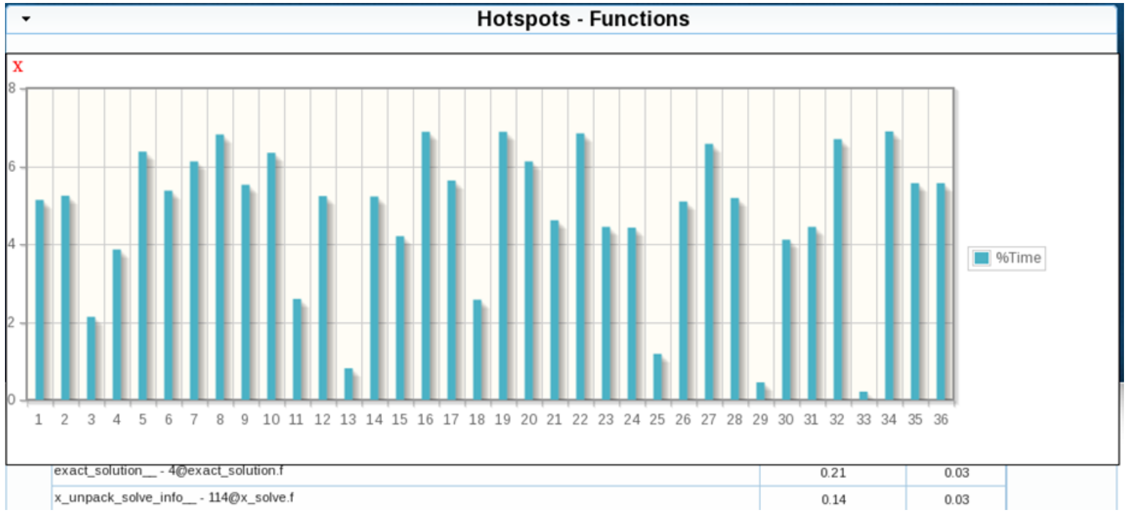
Introduction
MAQAO methodology

Decision tree



MAQAO LProf: Lightweight Profiler





MAQAO LProf: Lightweight Profiler
Introduction

Lightweight localization of application hotspots

Multiple measurement methods available:

- Sampling (default)
  - Hardware counters (through perf_event_open system call)
  - Non intrusive, low overhead
- Instrumentation
  - Through binary rewriting
  - For targeting specific issues
  - Extra overhead

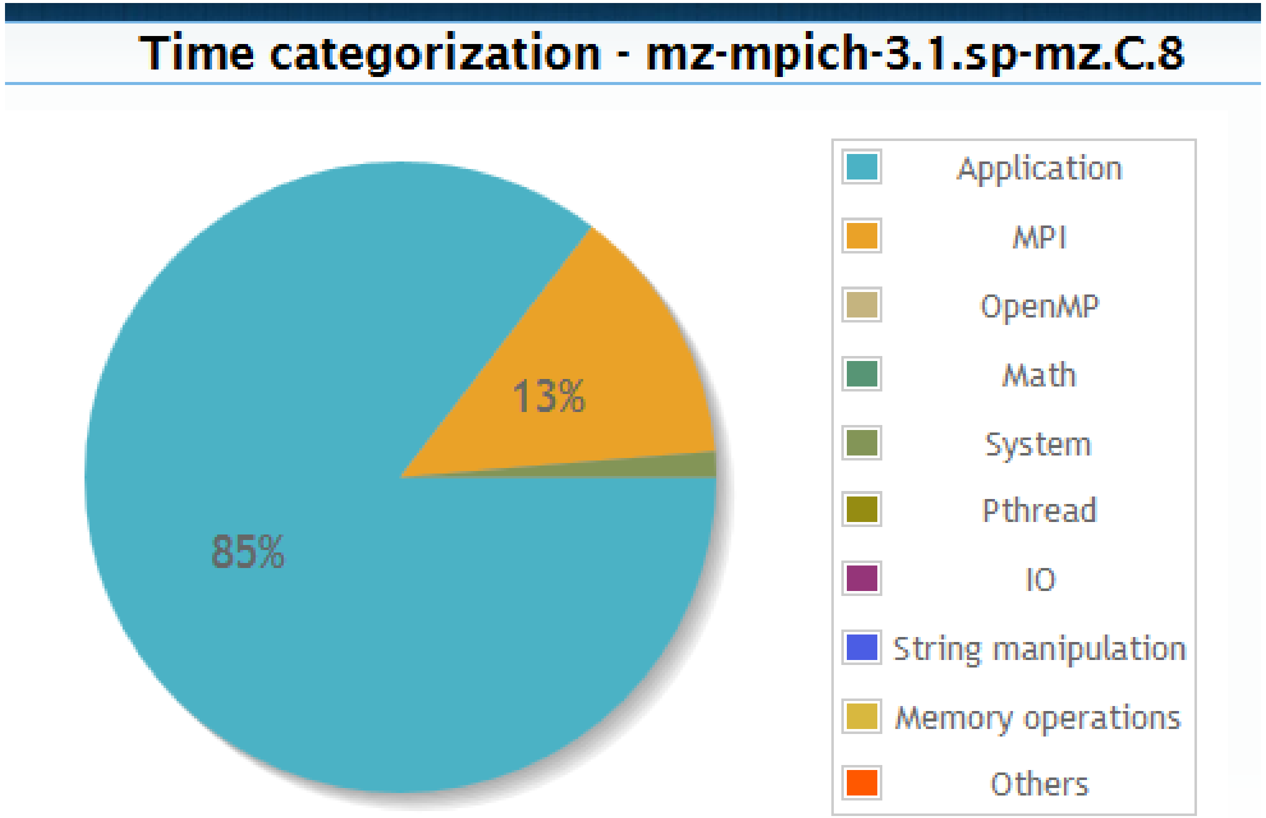
Runtime-agnostic

MAQAO LProf: Lightweight Profiler
Time categorization

Parallelization overhead:
- Shared: Pthreads, OpenMP, etc …
- Distributed: MPI, etc…

Programming:
- IO operations
- String operations
- Memory management
- External libraries such as libm / libmkl

User time breakdown:
- Functions
- Loops



MAQAO LProf: Lightweight Profiler
Function and loop hotspots (1/3)

Focusing on user time:

- Function hotspots
- Load balancing across the nodes

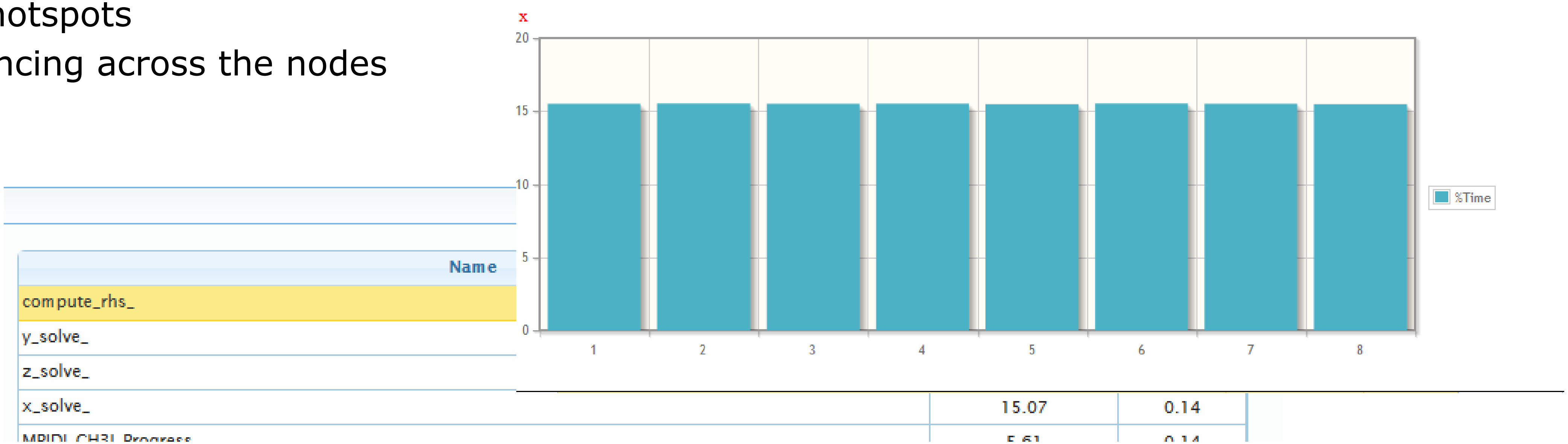
Hotspots - Functions

Name	Median Excl %Time	Deviation
compute_rhs_	30.88	0.14
y_solve_	15.51	0.14
z_solve_	15.34	0.14
x_solve_	15.07	0.14
MPIDI_CH3I_Progress	5.61	0.14

MAQAO LProf: Lightweight Profiler
Function and loop hotspots (2/3)

Focusing on user time:

- Function hotspots
- Load balancing across the nodes



MAQAO LProf: Lightweight Profiler
Function and loop hotspots (3/3)

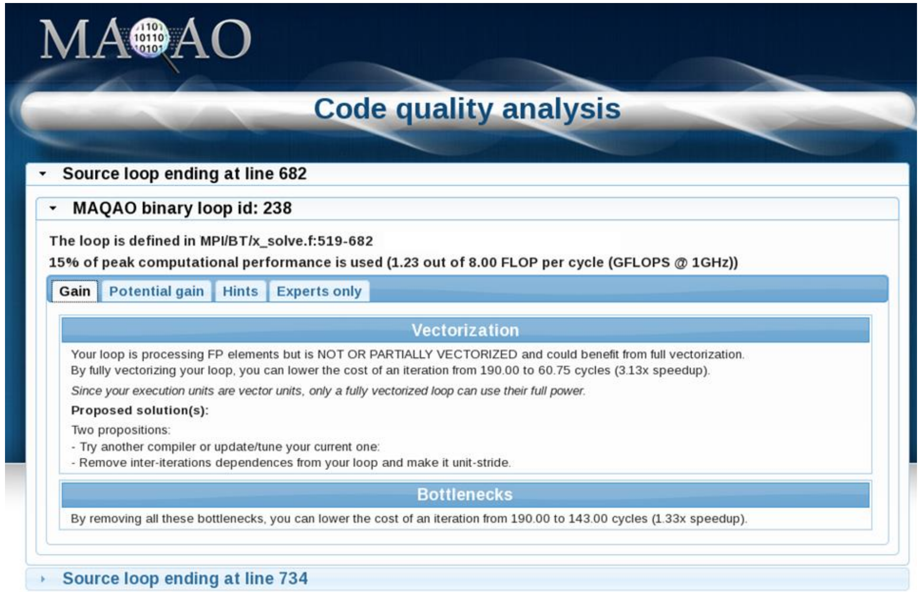
Analyzing the time spent at loop level:

- Finding the most time consuming
- Providing direct link to MAQAO CQA analyses

dauvergne - Process #14213 - Thread #14201

Name	Excl %Time	Excl Time (s)
binvcrhs - 206@solve_subs.f	17.27	2.23
MPIDI_CH3I_Progress	15.24	1.96
poll_active_fboxes	13.71	1.77
▾ y_solve_.omp_fn.0 - 45@y_solve.f	8.47	1.09
▾ loops	8.47	
▾ Loop 121 - y_solve.f@45	0	
▾ Loop 122 - y_solve.f@45	0.16	
○ Loop 124 - y_solve.f@45	0.14	
○ Loop 125 - y_solve.f@145	5.12	
○ Loop 126 - y_solve.f@55	2.03	
○ Loop 123 - y_solve.f@45	1.02	
▾ x_solve_.omp_fn.0 - 48@x_solve.f	8.23	1.06
▸ loops	8.23	

MAQAO CQA: Code Quality Analyzer



MAQAO CQA: Code Quality Analyzer
Introduction

Improving performance of the user code

Focusing on loops:

- In HPC most of the time is spent in loops

Performing static analysis of assembly code based on a microarchitecture model:

- Evaluates the quality of the compiler generated code
- Returns hints and workarounds to the developer

Targets compute bound codes

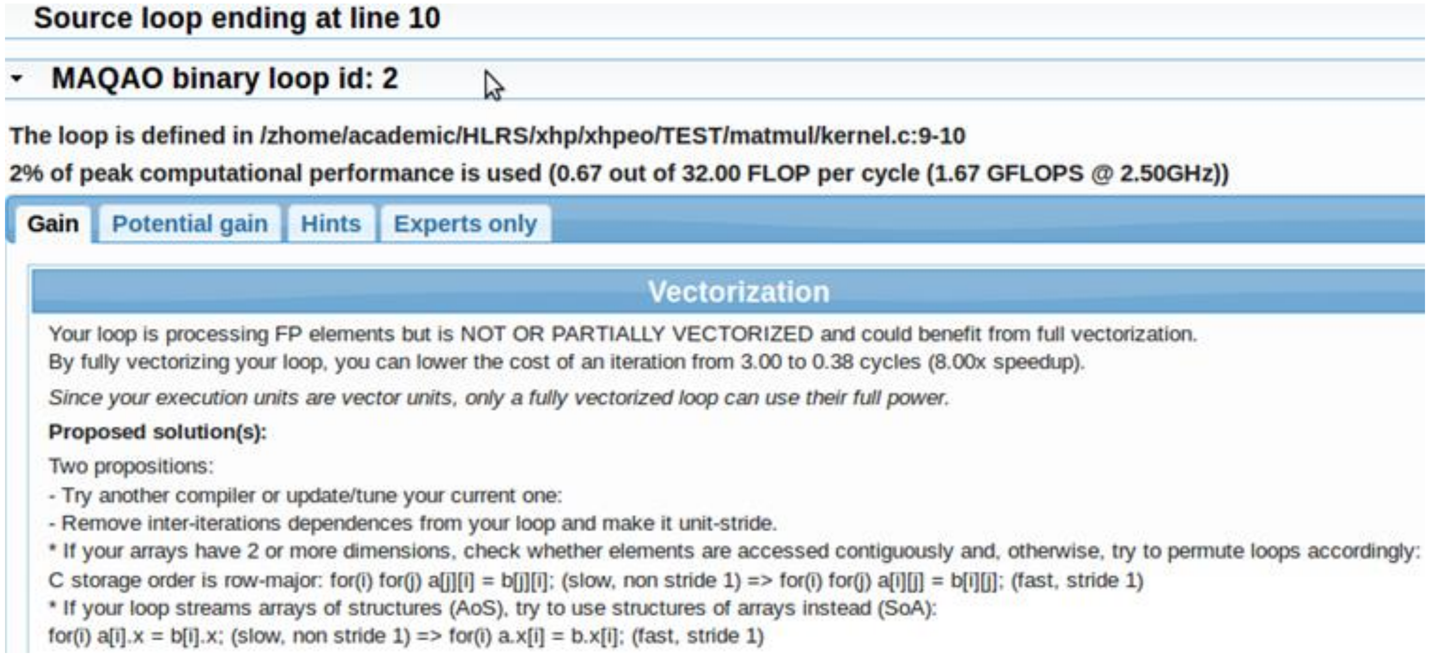
MAQAO CQA: Code Quality Analyzer
Output

High level reports:

- Reference to the source code
- Bottleneck description
- Hints to improve performance
- Reports categorized by confidence level
  - gain, potential gain

Low level report for performance experts

No runtime cost/overhead



MAQAO CQA: Code Quality Analyzer
Processor Architecture: Core level

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

Concepts:
- 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

Same instruction – Same cost

Process up to 8X (SP) data

MAQAO CQA: Code Quality Analyzer
Compiler and programmer hints

Compiler can be driven using flags and pragmas:

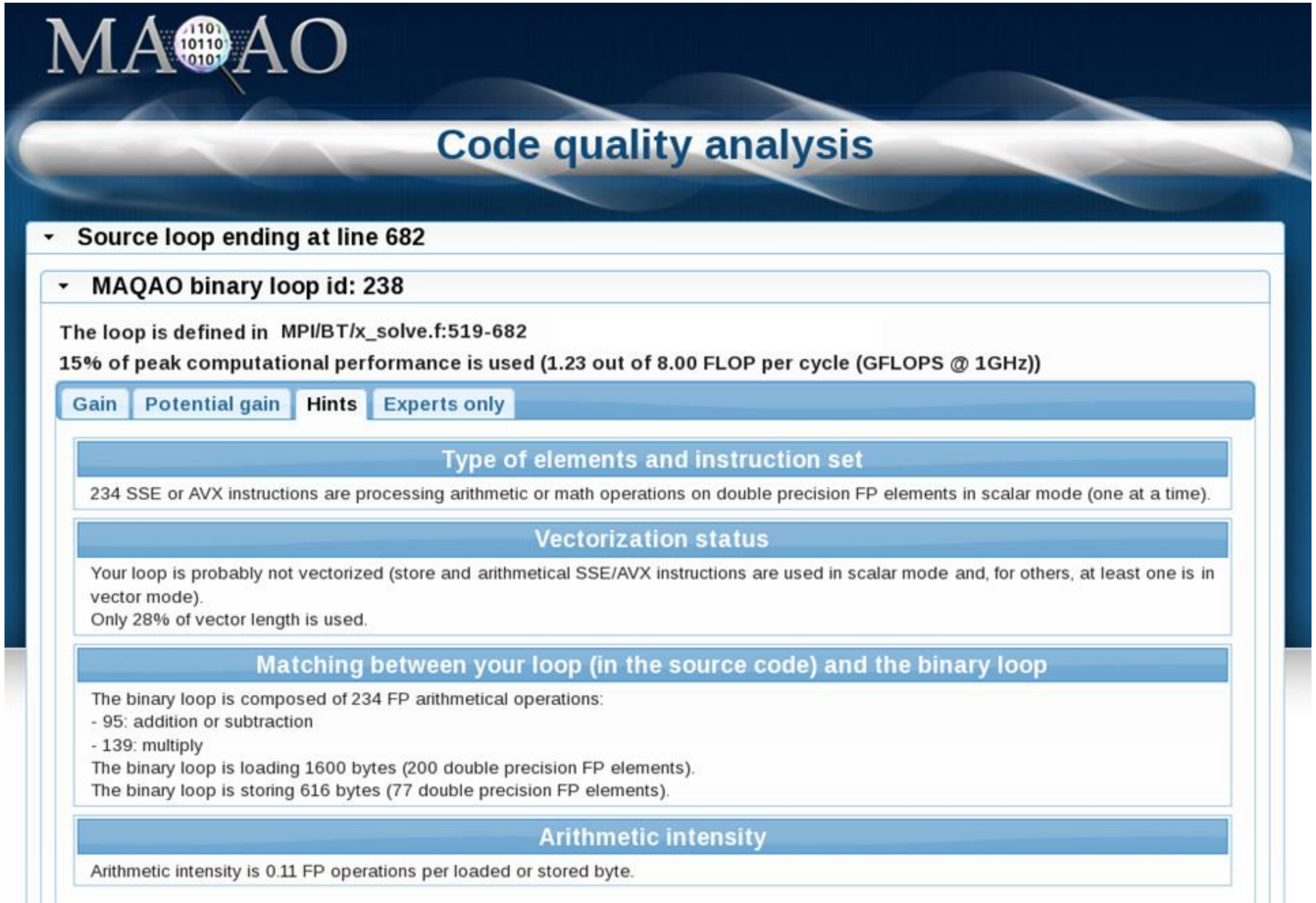
- Ensuring full use of architecture capabilities (e.g. using flag -xHost on AVX capable machines)
- Forcing optimization (unrolling, vectorization, alignment, …)
- Bypassing conservative behavior when possible (e.g. 1/X precision)

Implementation changes

- Improve data access
  - Loop interchange
  - Changing loop strides
- Avoid instructions with high latency

MAQAO CQA: Code Quality Analyzer
GUI sample (1/2)

MAQAO CQA: Code Quality Analyzer
GUI sample (2/2)



Thank you for your attention !

Questions ?