



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

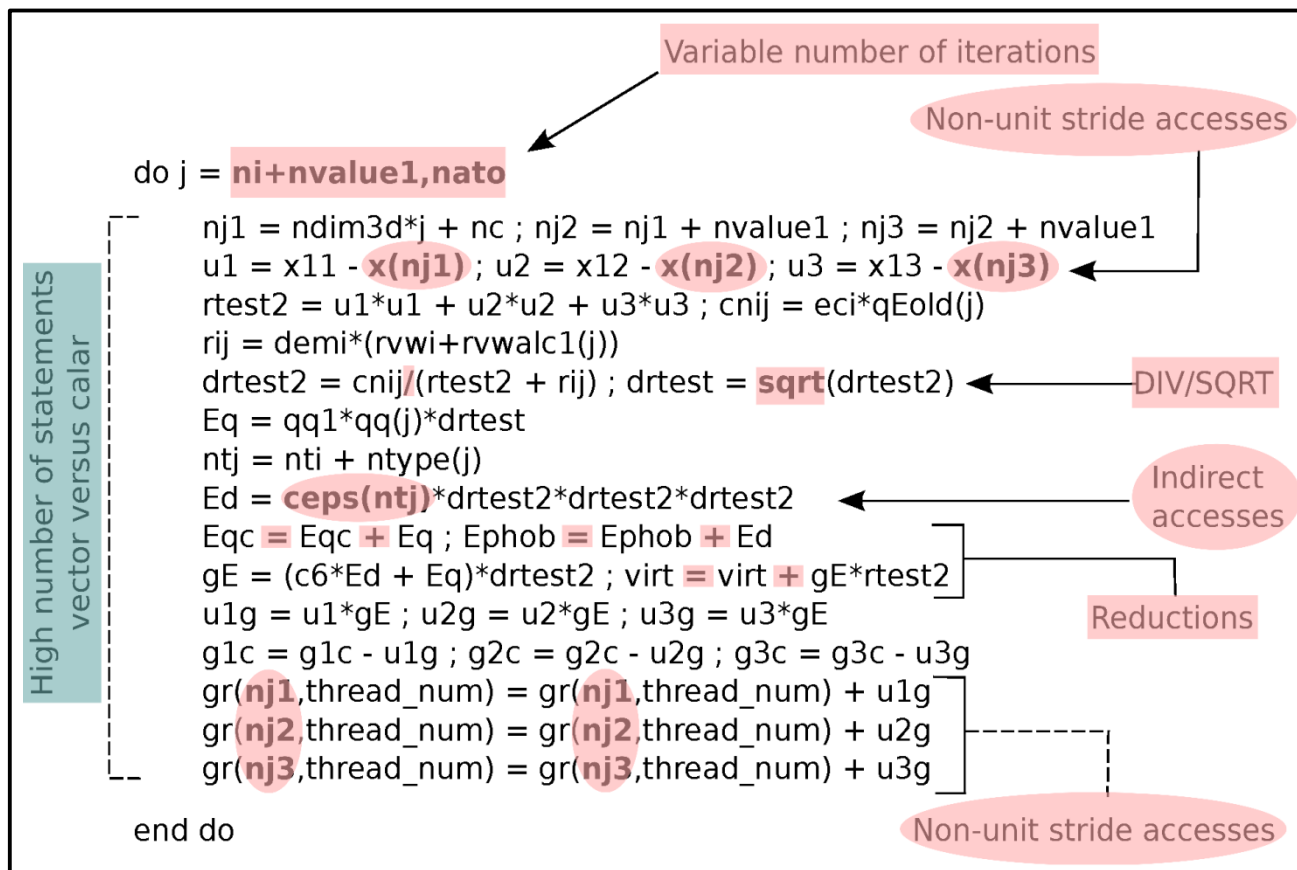
=> Need for dedicated and complementary tools



Introduction

Performance analysis (2/2)

Motivating example: loop ~10% walltime



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

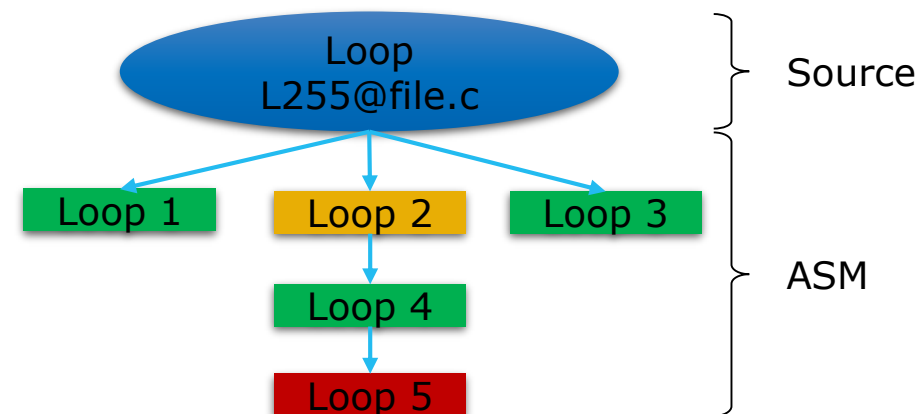
Advantages of binary analysis:

- 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

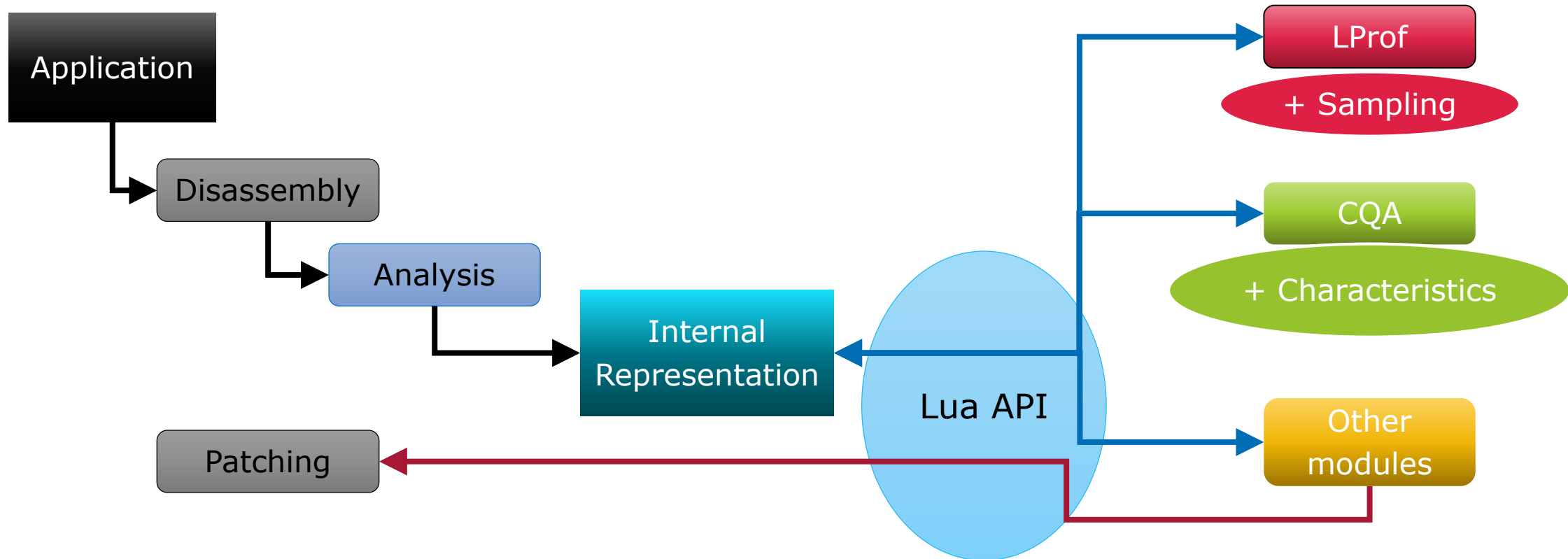
Main steps:

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



Introduction

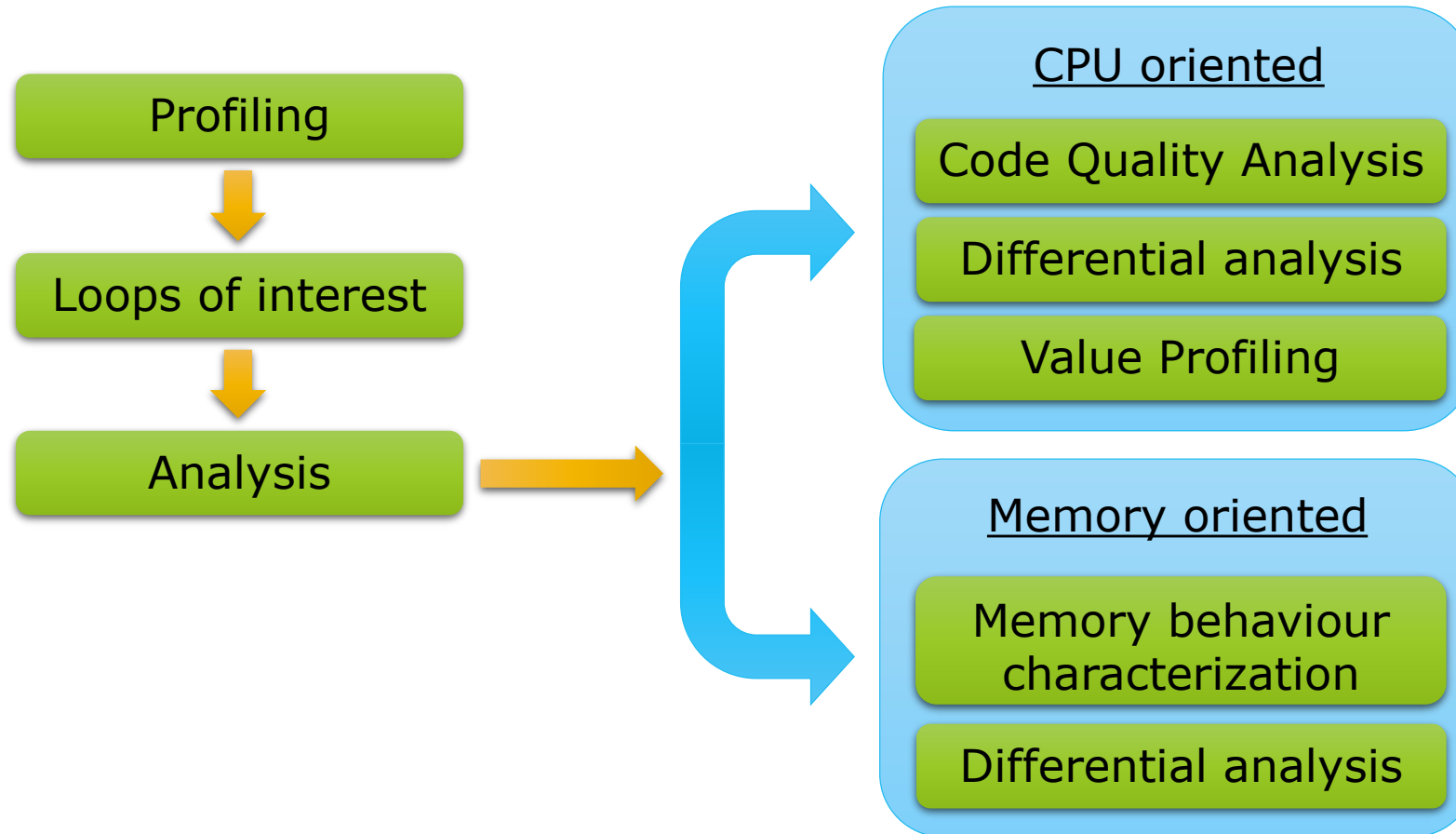
MAQAO Main Structure



Introduction

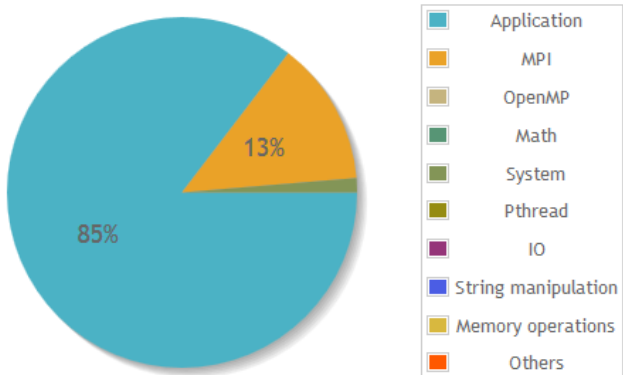
MAQAO methodology

Decision tree

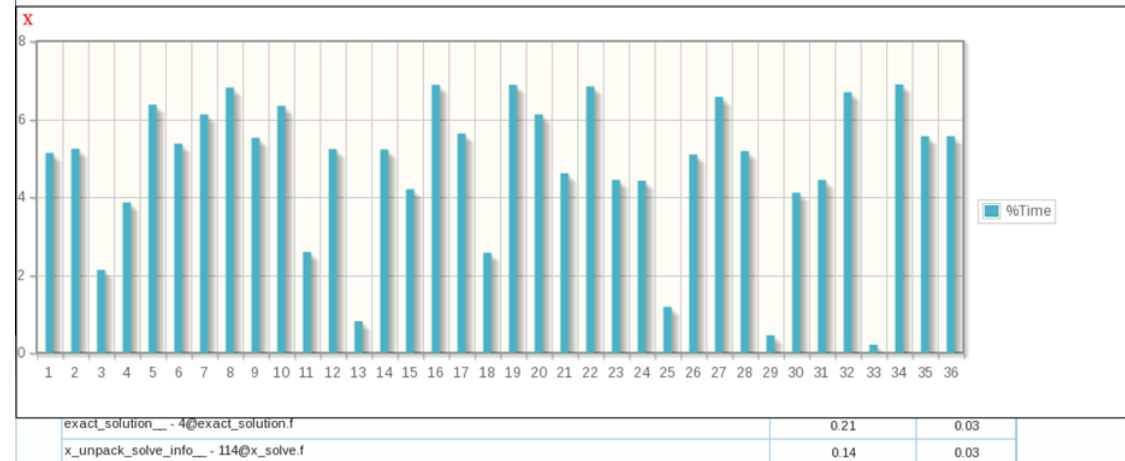


MAQAO LProf: Lightweight Profiler

Time categorization - mz-mpich-3.1.sp-mz.C.8



Hotspots - Functions



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...

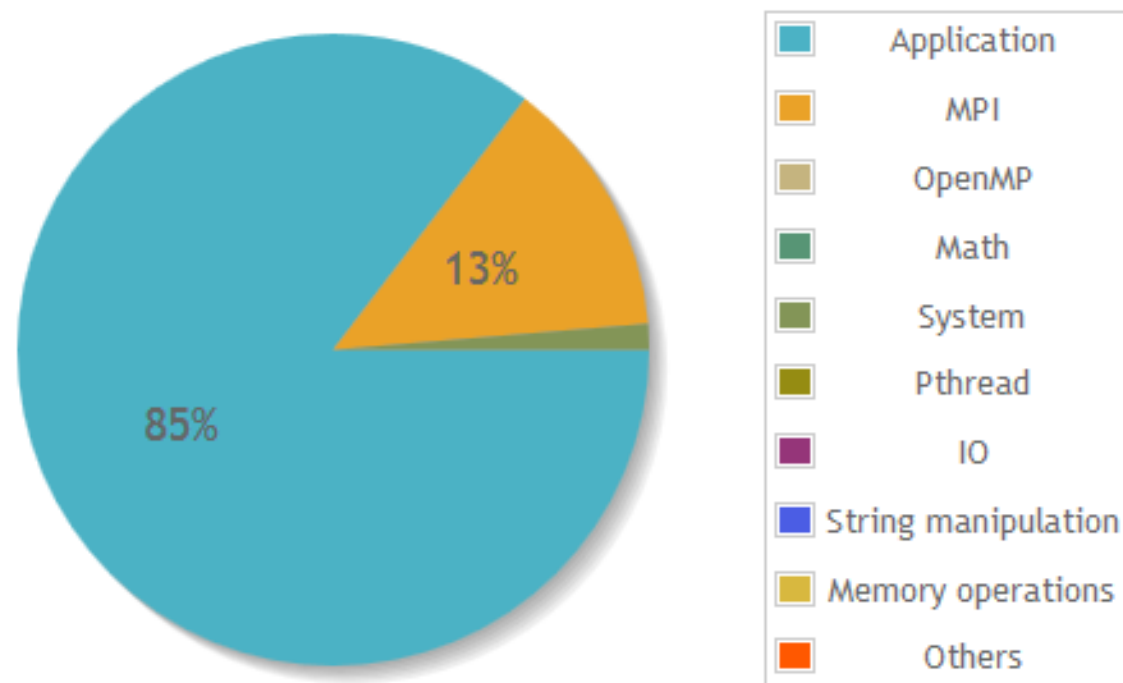
Programming:

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

User time breakdown:

- Functions
- Loops

Time categorization - mz-mpich-3.1.sp-mz.C.8



MAQAO LProf: Lightweight Profiler

Function and loop hotspots (1/3)

Focusing on user time:

- Function hotspots
- Load balancing across the nodes

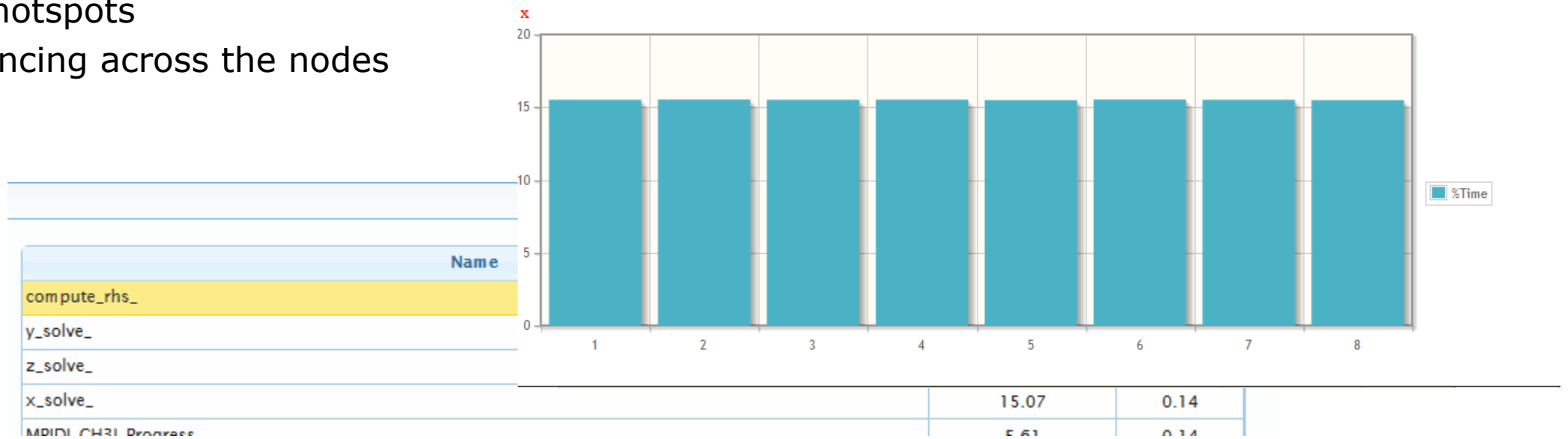
Hotspots - Functions		
Name	Median Excl %Time	Deviation
compute_rhs_	30.88	0.14
y_solve_	15.51	0.14
z_solve_	15.34	0.14
x_solve_	15.07	0.14
MDIOL CH3I Progress	5.61	0.14

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

The image shows a screenshot of the MAQAO (MAQAO Performance Analysis and Optimization Tool) Code Quality Analyzer (CQA) interface. The interface has a dark blue header with the MAQAO logo and the title "Code quality analysis". Below the header, there is a section titled "Source loop ending at line 682". Under this, a sub-section "MAQAO binary loop id: 238" is expanded, showing details about the loop's performance and optimization suggestions.

Source loop ending at line 682

MAQAO binary loop id: 238

The loop is defined in MPI/BT/x_solve.f:519-682
15% of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))

Gain Potential gain Hints Experts only

Vectorization

Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could benefit from full vectorization. By fully vectorizing your loop, you can lower the cost of an iteration from 190.00 to 60.75 cycles (3.13x speedup).
Since your execution units are vector units, only a fully vectorized loop can use their full power.

Proposed solution(s):

Two propositions:

- Try another compiler or update/tune your current one:
- Remove inter-iterations dependences from your loop and make it unit-stride.

Bottlenecks

By removing all these bottlenecks, you can lower the cost of an iteration from 190.00 to 143.00 cycles (1.33x speedup).

Source loop ending at line 734

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

Source loop ending at line 10

MAQAO binary loop id: 2

The loop is defined in /zhome/academic/HLRS/xhp/xhpeo/TEST/matmul/kernel.c:9-10
2% of peak computational performance is used (0.67 out of 32.00 FLOP per cycle (1.67 GFLOPS @ 2.50GHz))

Gain Potential gain Hints Experts only

Vectorization

Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could benefit from full vectorization. By fully vectorizing your loop, you can lower the cost of an iteration from 3.00 to 0.38 cycles (8.00x speedup).
Since your execution units are vector units, only a fully vectorized loop can use their full power.

Proposed solution(s):

Two propositions:

- Try another compiler or update/tune your current one:
- 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:
C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) => for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)

* If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA):
for(i) a[i].x = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)

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:

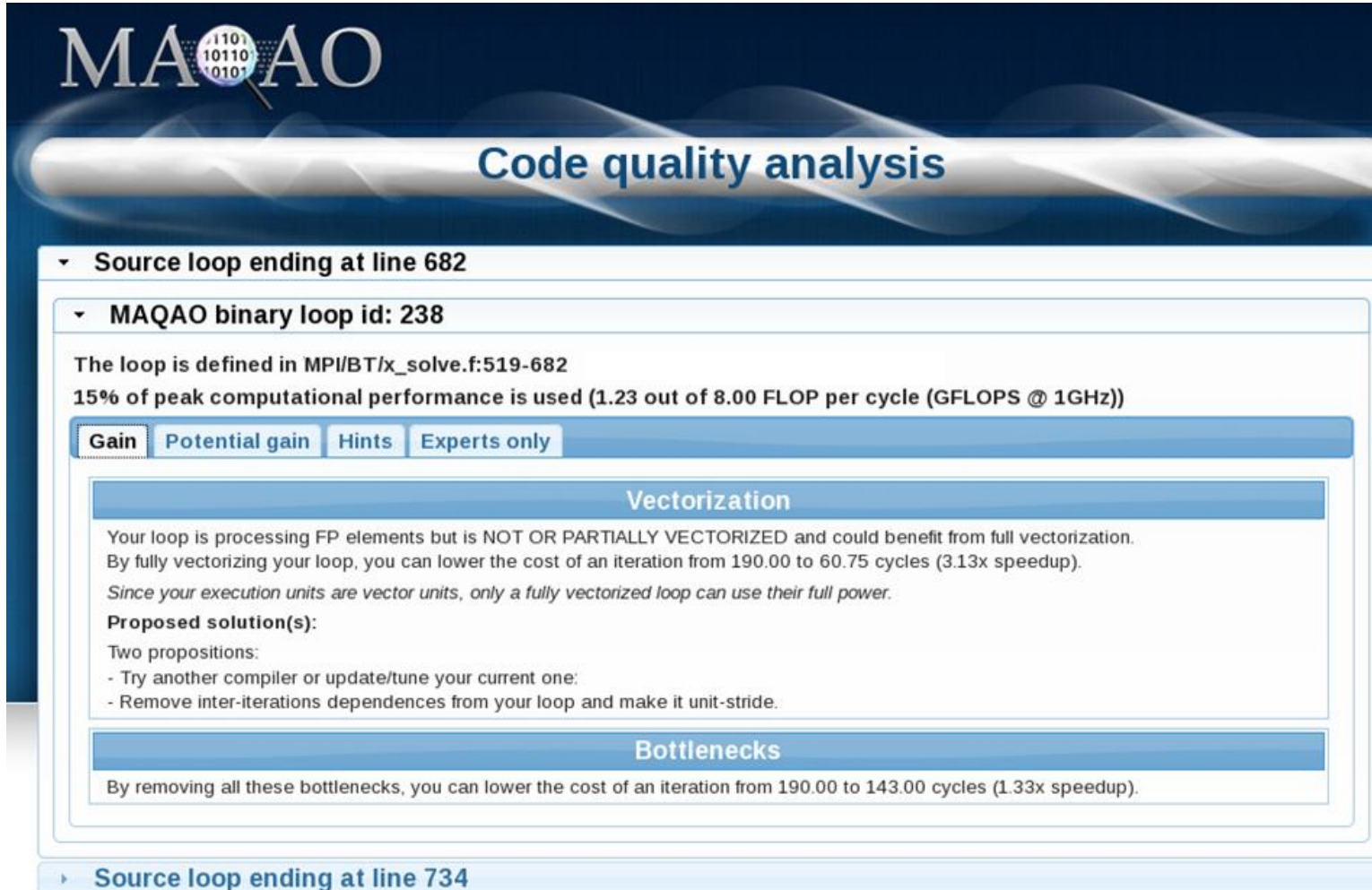
- Ensuring full use of architecture capabilities (e.g. using flag -xHost on AVX capable machines)
- Forcing optimization (unrolling, vectorization, alignment, ...)
- Bypassing conservative behavior when possible (e.g. 1/X precision)

Implementation changes

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

MAQAO CQA: Code Quality Analyzer

GUI sample (1/2)



The screenshot displays the MAQAO Code Quality Analyzer (CQA) interface. At the top, the MAQAO logo is shown next to the title "Code quality analysis". Below this, a dropdown menu indicates the "Source loop ending at line 682". Underneath, another dropdown shows the "MAQAO binary loop id: 238". The main text area provides details about the loop: "The loop is defined in MPI/BT/x_solve.f:519-682" and "15% of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))". A horizontal bar contains four tabs: "Gain" (selected), "Potential gain", "Hints", and "Experts only". The "Gain" tab is active, showing a section titled "Vectorization". This section contains the text: "Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could benefit from full vectorization. By fully vectorizing your loop, you can lower the cost of an iteration from 190.00 to 60.75 cycles (3.13x speedup). Since your execution units are vector units, only a fully vectorized loop can use their full power." Below this, it lists "Proposed solution(s):" with two propositions: "Try another compiler or update/tune your current one:" and "Remove inter-iterations dependences from your loop and make it unit-stride." A second section titled "Bottlenecks" follows, stating: "By removing all these bottlenecks, you can lower the cost of an iteration from 190.00 to 143.00 cycles (1.33x speedup)." At the bottom, a dropdown menu shows "Source loop ending at line 734".

MAQAO

Code quality analysis

▼ Source loop ending at line 682

▼ MAQAO binary loop id: 238

The loop is defined in MPI/BT/x_solve.f:519-682
15% of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))

Gain Potential gain Hints Experts only

Vectorization

Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could benefit from full vectorization. By fully vectorizing your loop, you can lower the cost of an iteration from 190.00 to 60.75 cycles (3.13x speedup). Since your execution units are vector units, only a fully vectorized loop can use their full power.

Proposed solution(s):

Two propositions:

- Try another compiler or update/tune your current one:
- Remove inter-iterations dependences from your loop and make it unit-stride.

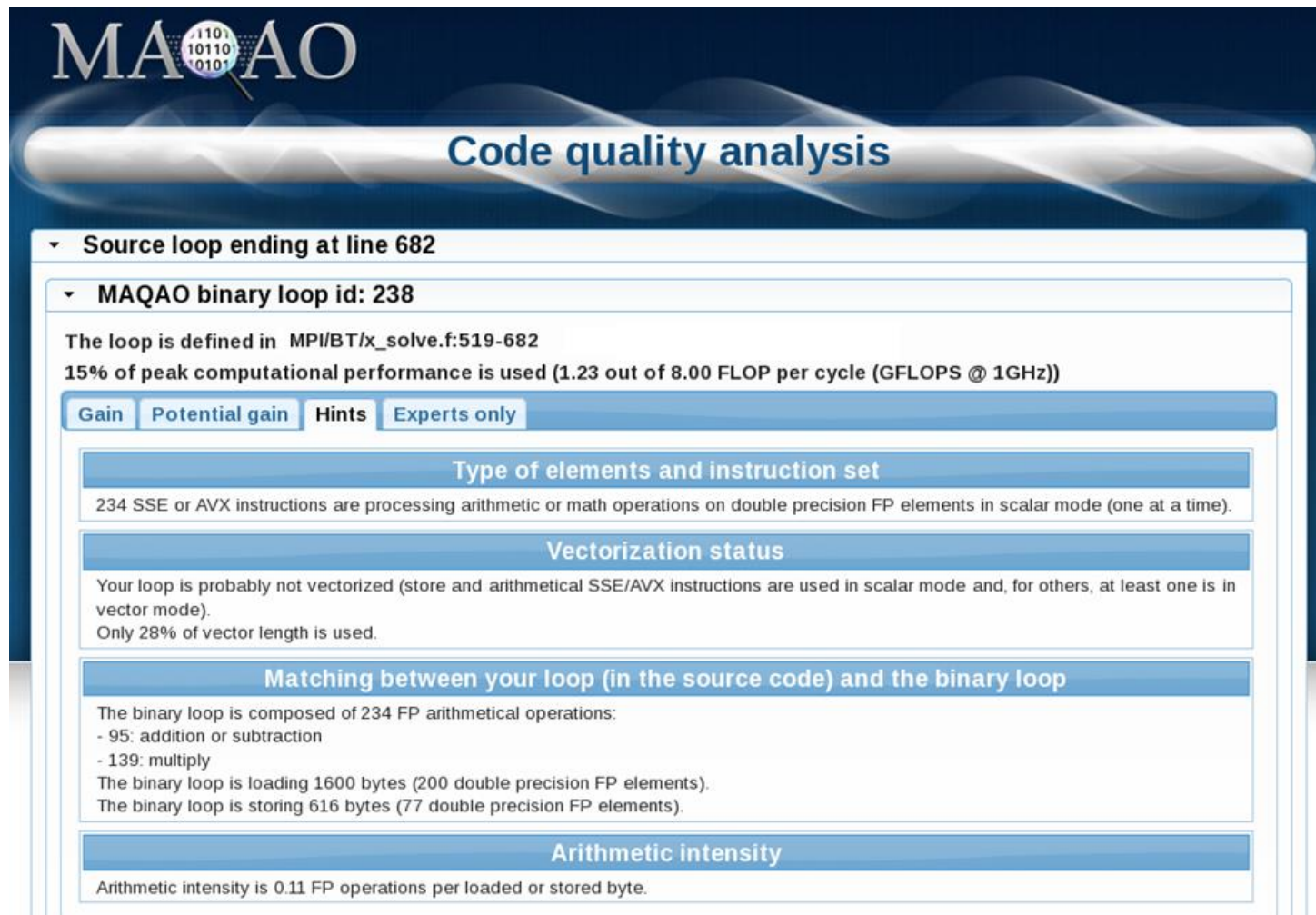
Bottlenecks

By removing all these bottlenecks, you can lower the cost of an iteration from 190.00 to 143.00 cycles (1.33x speedup).

► Source loop ending at line 734

MAQAO CQA: Code Quality Analyzer

GUI sample (2/2)



The screenshot displays the MAQAO Code Quality Analyzer (CQA) interface. At the top, the MAQAO logo is shown next to the title "Code quality analysis". Below this, a tree view shows the analysis path: "Source loop ending at line 682" > "MAQAO binary loop id: 238". The main content area provides details for this loop, including its definition in "MPI/BT/x_solve.f:519-682" and its performance metrics: "15% of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))". A tabbed interface allows switching between "Gain", "Potential gain", "Hints", and "Experts only". The "Hints" tab is currently active, showing four sections: "Type of elements and instruction set" (234 SSE or AVX instructions in scalar mode), "Vectorization status" (loop not vectorized, 28% of vector length used), "Matching between your loop (in the source code) and the binary loop" (234 FP operations: 95 additions/subtractions, 139 multiplies; 1600 bytes loaded, 616 bytes stored), and "Arithmetic intensity" (0.11 FP operations per byte).

MAQAO Code quality analysis

▼ Source loop ending at line 682

▼ MAQAO binary loop id: 238

The loop is defined in MPI/BT/x_solve.f:519-682
15% of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))

Gain Potential gain **Hints** Experts only

Type of elements and instruction set

234 SSE or AVX instructions are processing arithmetic or math operations on double precision FP elements in scalar mode (one at a time).

Vectorization status

Your loop is probably not vectorized (store and arithmetical SSE/AVX instructions are used in scalar mode and, for others, at least one is in vector mode).
Only 28% of vector length is used.

Matching between your loop (in the source code) and the binary loop

The binary loop is composed of 234 FP arithmetical operations:
- 95: addition or subtraction
- 139: multiply
The binary loop is loading 1600 bytes (200 double precision FP elements).
The binary loop is storing 616 bytes (77 double precision FP elements).

Arithmetic intensity

Arithmetic intensity is 0.11 FP operations per loaded or stored byte.

Thank you for your attention !

Questions ?