



MAQAO

Performance Analysis and Optimization Tool

Cédric VALENSI, Emmanuel OSERET, Mathieu TRIBALAT
{cedric.valensi, emmanuel.oseret, mathieu.tribalat}@uvsq.fr

Performance Evaluation Team, University of Versailles S-Q-Y

Andres S. CHARIF RUBIAL - ascr@pexl.eu - PeXL

<http://www.maqao.org>

VI-HPS 25th Aachen – Germany – 27-31 March 2017



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:

- 1) High number of statements (points to the loop body)
- 2) Non-unit stride accesses (points to `ni + nvalue1, nato` and the `gr` array accesses)
- 3) Indirect accesses (points to `ceps(ntj)`)
- 4) DIV/SQRT (points to `drtest = sqrt(drtest2)`)
- 5) Reductions (points to `Eqc = Eqc + Eq` and `Ephob = Ephob + Ed`)
- 6) Variable number of iterations (points to the `do` loop header)

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

Introduction

MAQAO: Modular Assembly Quality Analyzer and Optimizer

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...)



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

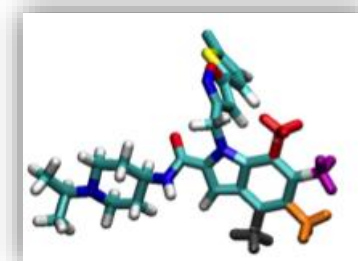
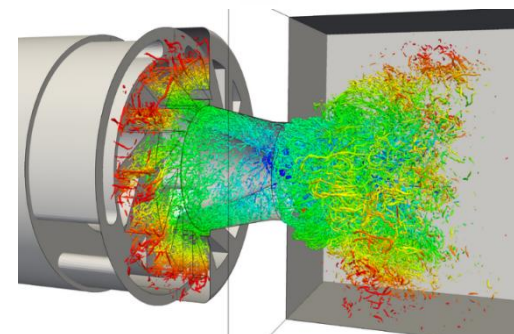
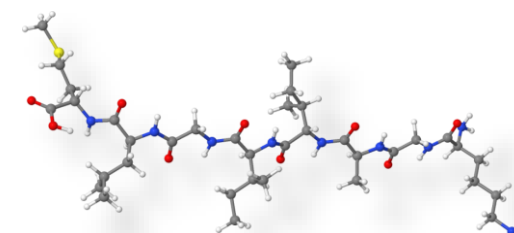


Introduction

Success stories

MAQAO was used for optimizing industrial and academic HPC applications:

- QMC=CHEM (IRSAMC)
 - Quantum chemistry
 - Speedup: > 3x
- Yales2 (CORIA)
 - Computational fluid dynamics
 - Speedup: up to 2,8x
- Polaris (CEA)
 - Molecular dynamics
 - Speedup: 1,5x – 1,7x
- AVBP (CERFACS)
 - Computational fluid dynamics
 - Speedup: 1,08x – 1,17x



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

MAQAO: Analysis at binary level

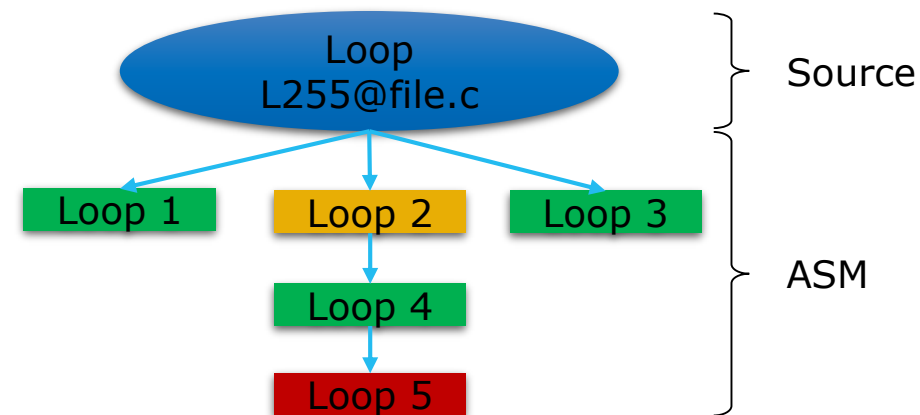
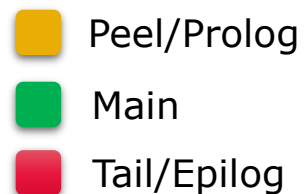
Advantages of binary analysis:

- 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

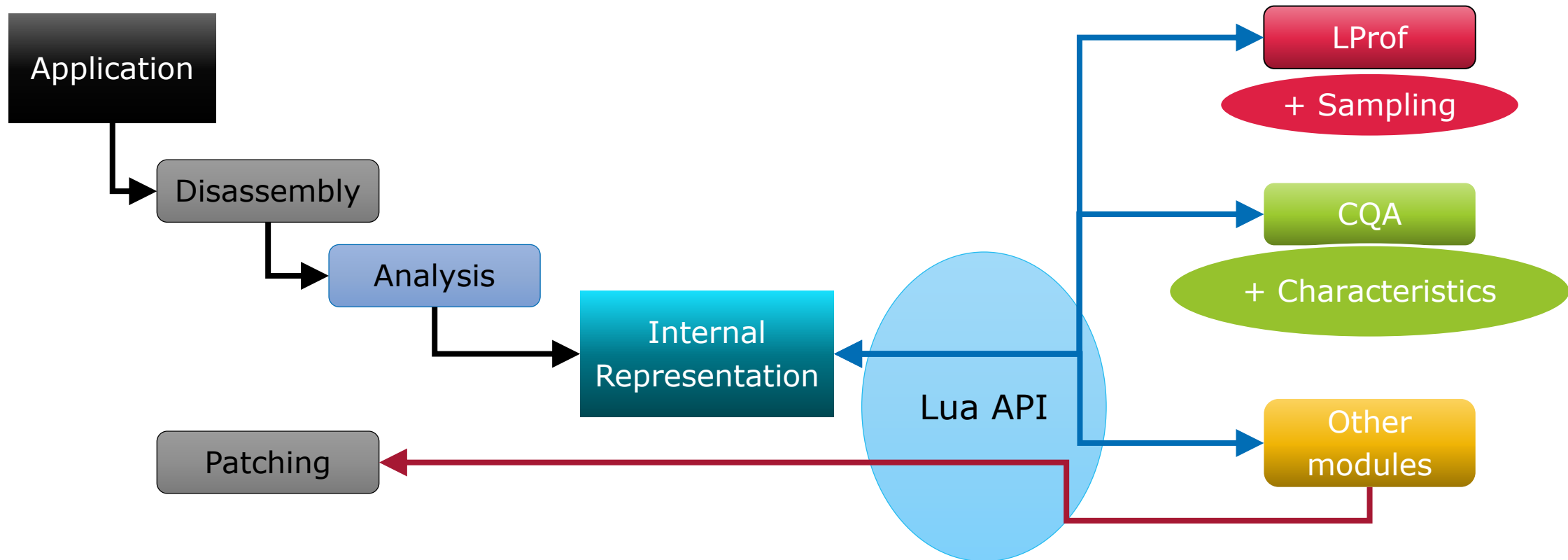
Main steps:

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



Introduction

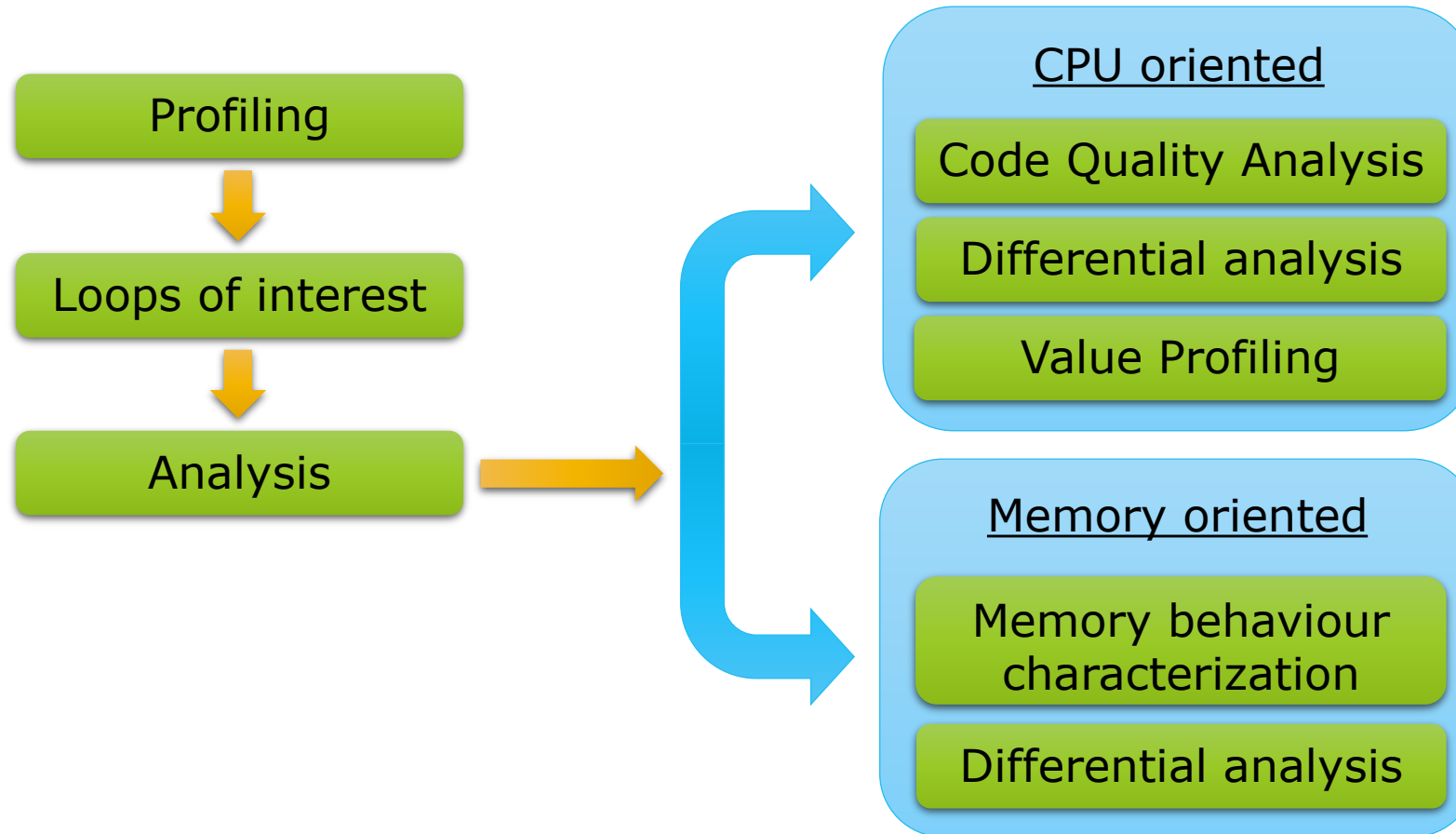
MAQAO Main structure



Introduction

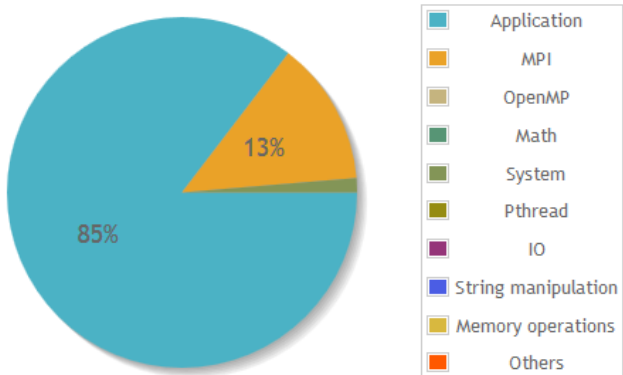
MAQAO Methodology

Decision tree

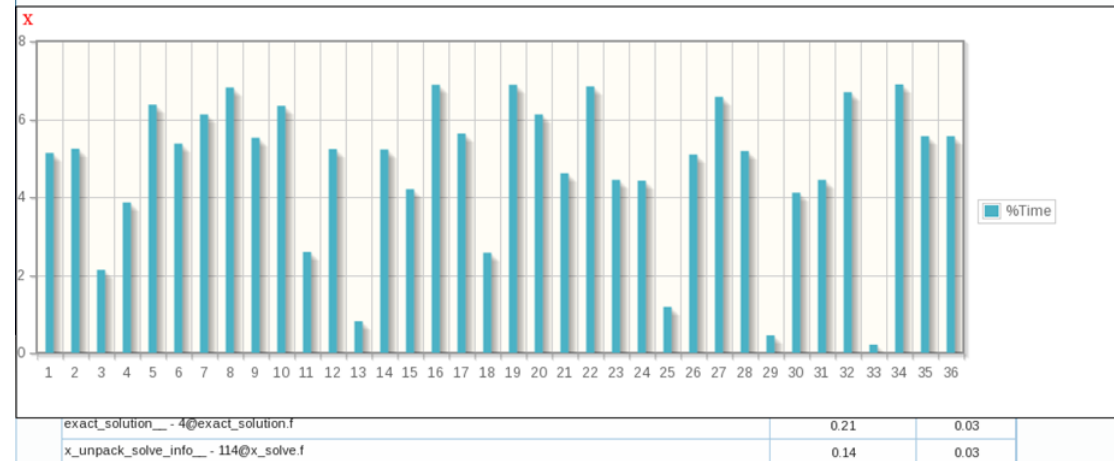


MAQAO LProf: Lightweight Profiler

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: for targeting specific issues
 - Binary rewriting
 - Extra overhead

Runtime-agnostic

MAQAO LProf: Lightweight Profiler

Time categorization

Parallelization overhead:

- Shared: Pthreads, OpenMP, etc ...
- Distributed: MPI, etc...

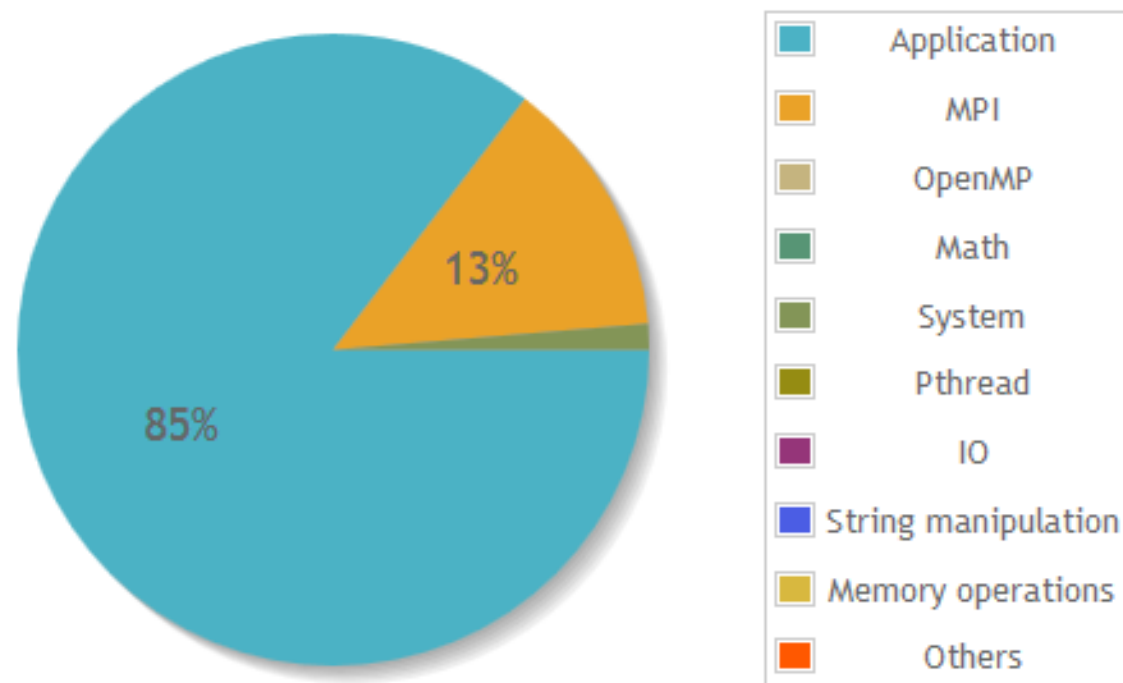
Programming:

- 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

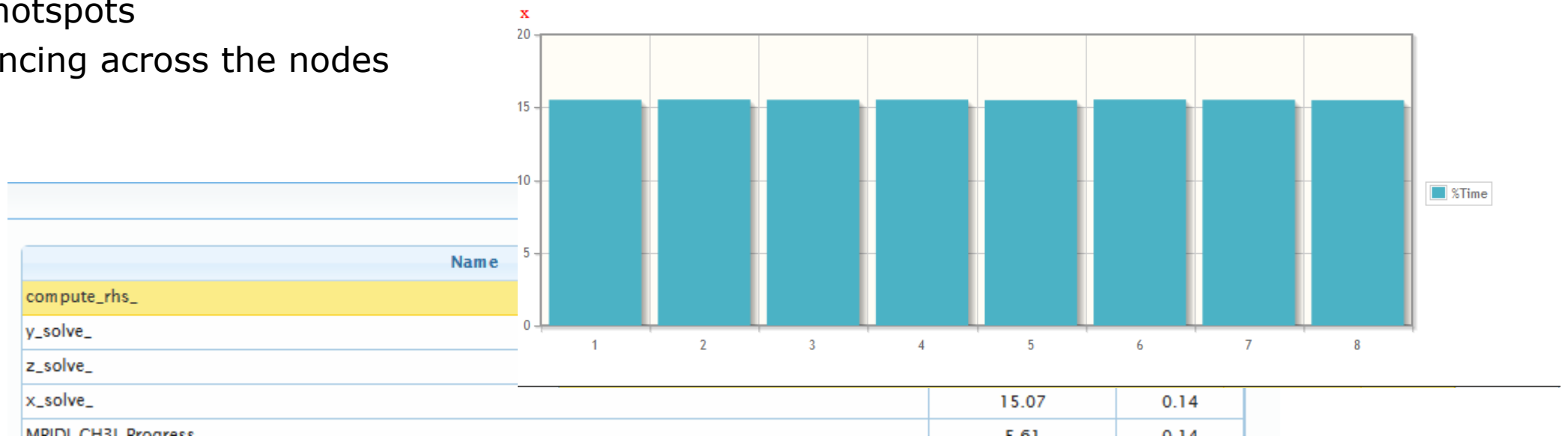
Hotspots - Functions		
Name	Median Excl %Time	Deviation
compute_rhs_	30.88	0.14
y_solve_	15.51	0.14
z_solve_	15.34	0.14
x_solve_	15.07	0.14
MDIOL CH31 Progress	5.61	0.14

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 Code Quality Analyzer (CQA) interface. The header features the MAQAO logo and the title "Code quality analysis". Below this, a dropdown menu shows "Source loop ending at line 682". Underneath, another dropdown shows "MAQAO binary loop id: 238". A text box states: "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))". There are four tabs: "Gain", "Potential gain", "Hints", and "Experts only". The "Gain" tab is selected, showing a section titled "Vectorization". The text in this section says: "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 says "Proposed solution(s):" and lists two propositions: "Try another compiler or update/tune your current one:" and "Remove inter-iterations dependences from your loop and make it unit-stride." There is also a section titled "Bottlenecks" which states: "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

Introduction

Improving performance of the user code

Performing static analysis of assembly code (no execution needed)

- Relies on a microarchitecture model
- Evaluates the quality of the compiler generated code
- Returns hints and workarounds to the developer

Focusing on loops:

- In HPC most of the time is spent in loops

Targets compute bound codes

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 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

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)

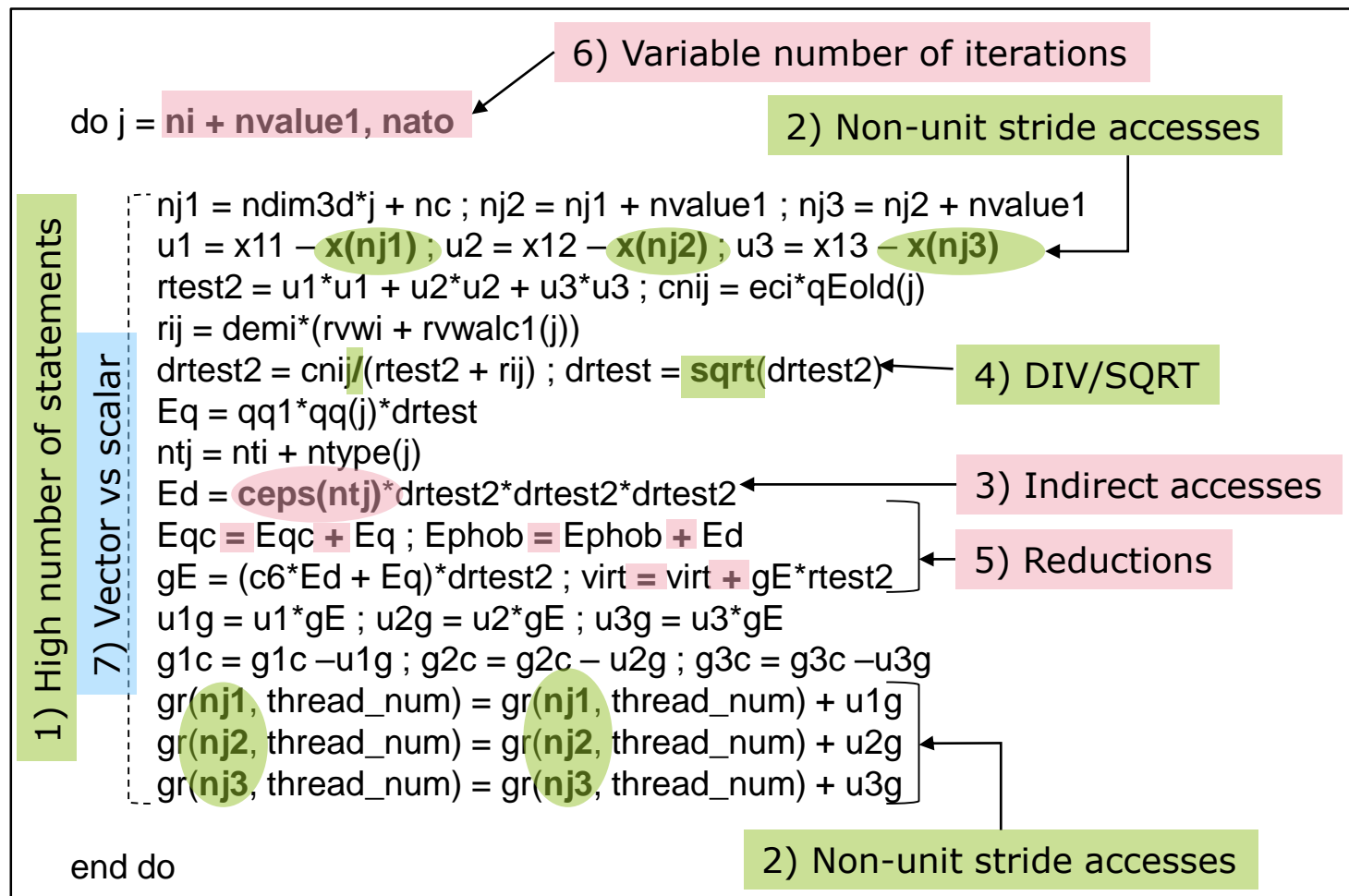
Implementation changes

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

MAQAO CQA: Code Quality Analyzer

Application to motivating example

Issues identified by CQA



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

MAQAO ONE View: Performance View Aggregator

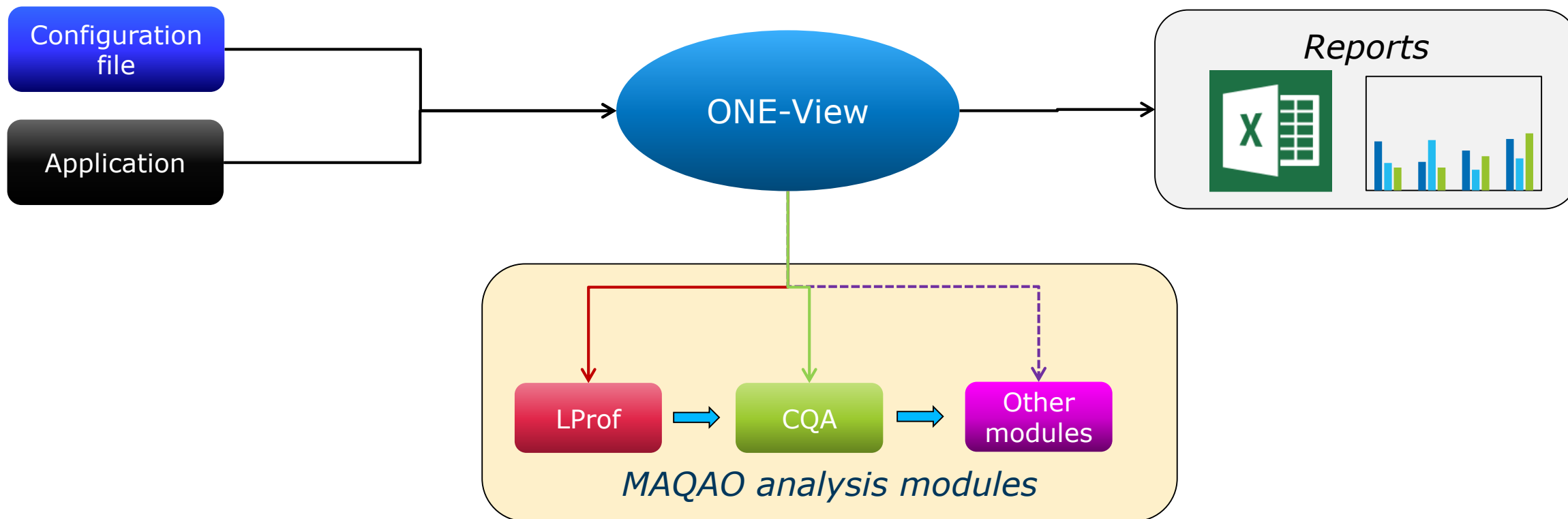


MAQAO ONE View: Performance View Aggregator

Introduction

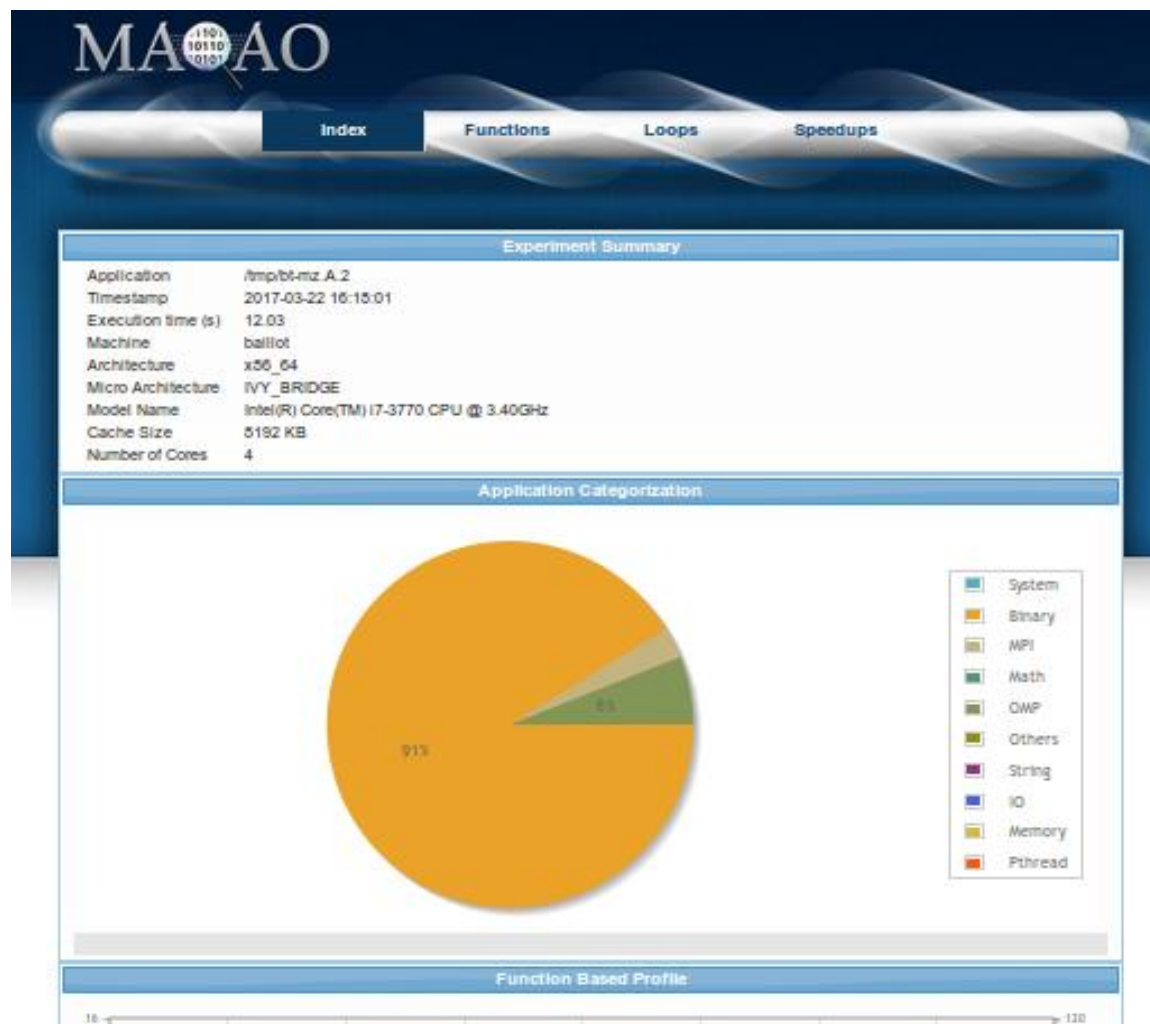
Automatizing the full analysis process

- Invocation of the MAQAO modules
- Generation of aggregated performance views as HTML or XLS graphs



MAQAO ONE View: Performance View Aggregator

GUI sample (1/4)



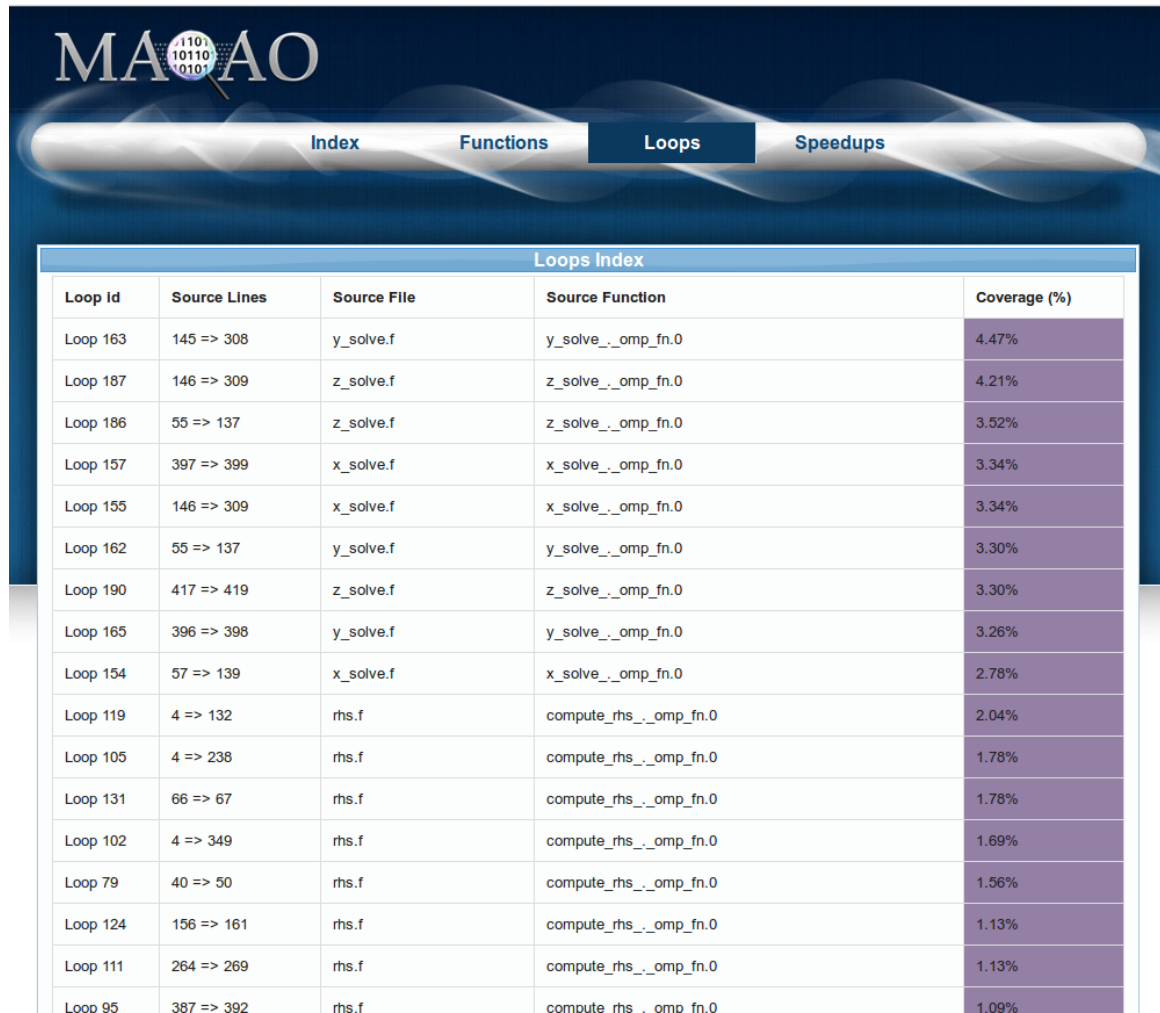
MAQAO ONE View: Performance View Aggregator

GUI sample (2/4)



The screenshot shows the MAQAO ONE View interface with the 'Functions' tab selected. The 'Functions and Loops' table displays a hierarchical view of performance data. The 'binvcrhs' function is expanded, showing a tree of loops with their respective coverage, thread counts, and deviations. Other functions like 'matmul_sub', 'omp_get_num_procs', and 'matvec_sub' are also listed.

Name	Coverage (%)	Nb Threads	Deviation
binvcrhs	24.13	4	2.31
compute_rhs__omp_fn.0	13.93	4	1.63
Loop 130 - rhs.f:4-178	3.43		
Loop 120 - rhs.f:4-132	2.04		
Loop 119 - rhs.f:4-132	2.04		
Loop 126 - rhs.f:155-161	1.13		
Loop 123 - rhs.f:139-151	0.17		
Loop 129 - rhs.f:166-178	0.09		
Loop 118 - rhs.f:4-288	3.12		
Loop 134 - rhs.f:64-67	1.78		
Loop 104 - rhs.f:4-349	1.69		
Loop 81 - rhs.f:39-50	1.56		
Loop 98 - rhs.f:386-392	1.09		
Loop 88 - rhs.f:430-433	0.26		
Loop 94 - rhs.f:402-406	0.13		
Loop 91 - rhs.f:415-419	0.09		
z_solve__omp_fn.0	12.63	4	0.89
y_solve__omp_fn.0	11.94	4	0.83
matmul_sub	11.41	4	0.7
x_solve__omp_fn.0	10.42	4	0.88
omp_get_num_procs	5.86	4	0.51
matvec_sub	3.78	4	0.2
MPIDI_CH3_iStartMsgv	1.26	2	0.44
add__omp_fn.0	0.82	4	0.19
MPIDI_CH3l_Progress	0.69	2	0.01
lhsinit	0.65	4	0.33
binvrhs	0.52	4	0.36



The screenshot shows the MAQAO ONE View interface with the 'Loops' tab selected. The 'Loops Index' table provides a detailed view of individual loops, including their IDs, source line ranges, source files, source functions, and coverage percentages.

Loop Id	Source Lines	Source File	Source Function	Coverage (%)
Loop 163	145 => 308	y_solve.f	y_solve__omp_fn.0	4.47%
Loop 187	146 => 309	z_solve.f	z_solve__omp_fn.0	4.21%
Loop 186	55 => 137	z_solve.f	z_solve__omp_fn.0	3.52%
Loop 157	397 => 399	x_solve.f	x_solve__omp_fn.0	3.34%
Loop 155	146 => 309	x_solve.f	x_solve__omp_fn.0	3.34%
Loop 162	55 => 137	y_solve.f	y_solve__omp_fn.0	3.30%
Loop 190	417 => 419	z_solve.f	z_solve__omp_fn.0	3.30%
Loop 165	396 => 398	y_solve.f	y_solve__omp_fn.0	3.26%
Loop 154	57 => 139	x_solve.f	x_solve__omp_fn.0	2.78%
Loop 119	4 => 132	rhs.f	compute_rhs__omp_fn.0	2.04%
Loop 105	4 => 238	rhs.f	compute_rhs__omp_fn.0	1.78%
Loop 131	66 => 67	rhs.f	compute_rhs__omp_fn.0	1.78%
Loop 102	4 => 349	rhs.f	compute_rhs__omp_fn.0	1.69%
Loop 79	40 => 50	rhs.f	compute_rhs__omp_fn.0	1.56%
Loop 124	156 => 161	rhs.f	compute_rhs__omp_fn.0	1.13%
Loop 111	264 => 269	rhs.f	compute_rhs__omp_fn.0	1.13%
Loop 95	387 => 392	rhs.f	compute_rhs__omp_fn.0	1.09%

MAQAO ONE View: Performance View Aggregator

GUI sample (3/4)

MAQAO

Index Functions Loops Speedups

Loop 119

Coverage	2.04 %
Function	compute_rhs__omp_fn.0
Source lines and file	4,132@rhs.f

Source Code
Assembly Code

Static Reports

CQA Report

The loop is defined in /home/cvalensi/Documents/Maqao/Tests/samples/NPB3.3.1-MZ/NPB3.3-MZ-MPI/BT-MZ/rhs.f:4-132
In the binary file, the address of the loop is: 406bbf

Path 1

19% of peak computational performance is used (1.54 out of 8.00 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 61.00 to 58.00 cycles (1.05x speedup).

Workaround

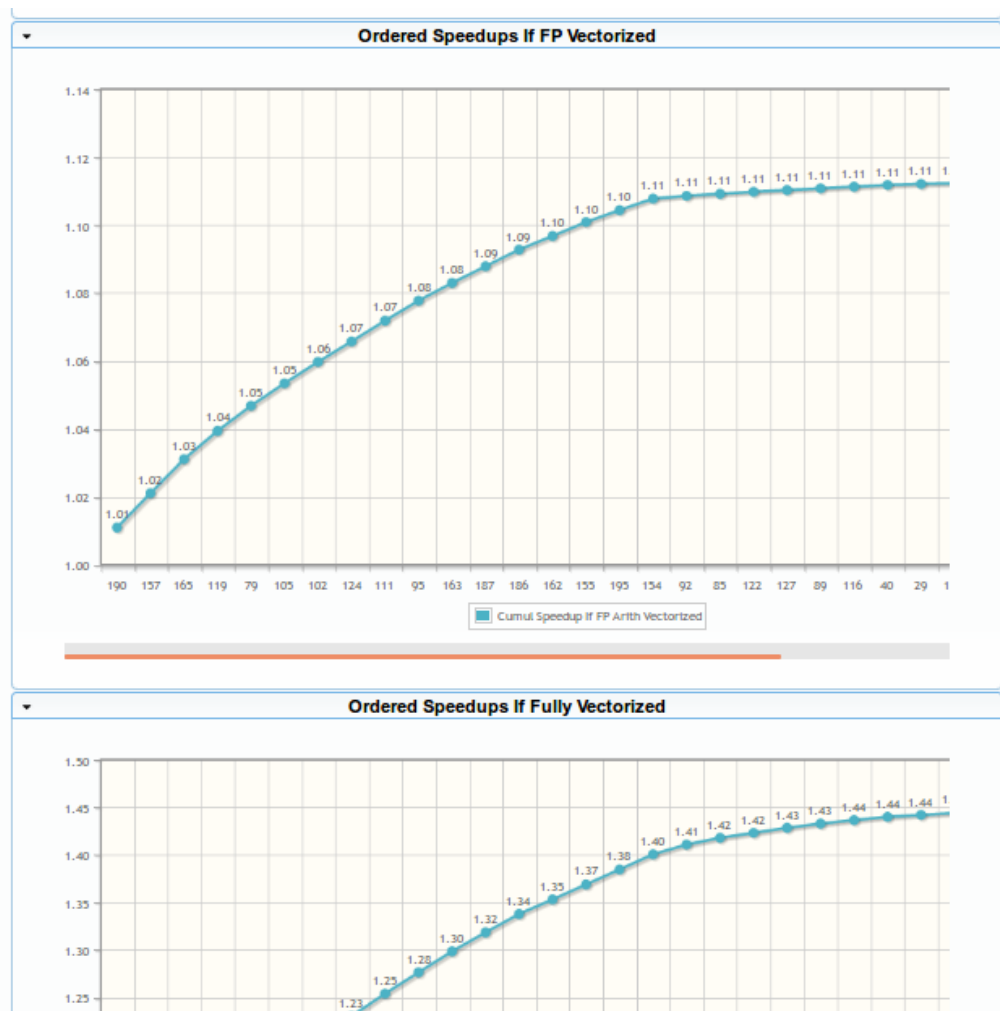
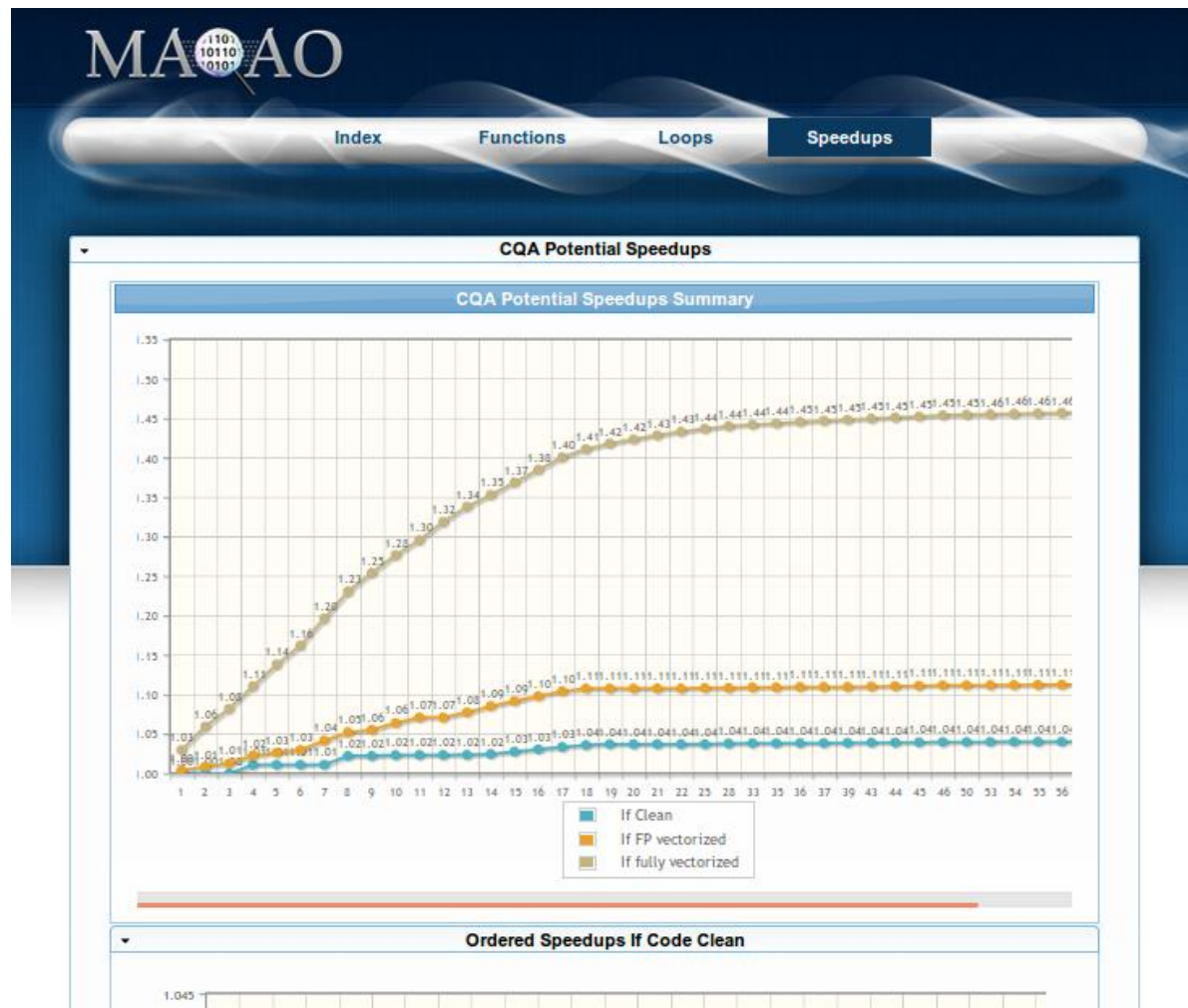
To reference allocatable arrays, use "allocatable" instead of "pointer" pointers or qualify them with the "contiguous" attribute (Fortran 2008). For structures, limit to one indirection. For example, use a_b%c instead of a%b%c with a_b set to a%b before this loop.

Vectorization status

Your loop is probably not vectorized (store and arithmetical SSE/AVX instructions are used in scalar mode and for

MAQAO ONE View: Performance View Aggregator

GUI sample (4/4)



Thank you for your attention !

Questions ?