Performance Evaluation MAQAO Toolsuite

Ter@tec – 2nd July 2014

Andrés S. Charif-Rubial, William Jalby

Outline

- Characterize the performance of an application
  - Complex multicore CPUs and memory systems
  - How well does it behaves on a given machine

- Generally a multifaceted problem
  - What are the issues (numerous but finite) ?
  - Which one(s) dominates ?
  - Maximizing the number of views
  - **=> Need for specialized tools**

- Several tools available
  - Which one to use ?
  - **=> Need for a methodology**

- ROI-oriented and global view:
  - Lack of performance impact prediction:
    => Will fixing a given pathology pay off ?
    => No way to get a return on investment metric
  - Global view:
    => what are the issues
    => which one has a high level speedup potential
  - Can lead to useless optimization:
    - Example 1: restructuring data accesses across all the application may be a loss of time if the potential speedup is only 2%
    - Example 2: various tools can detect high miss rates. It can be useless to fix a high miss rate if combined with div/sqrt operations because the dominating bottleneck might be FP operations.

- One-way approaches/techniques:

  - HPCToolKit, PerfExpert, VTune heavily rely on sampling and hardware events.
    => Sampling-based profiling aggregates everything together (all instances): might be counterproductive

  - Scalasca/Vampir is heavily relying on tracing and source code probe insertion
    => Tracing-based profiling is heavier (time consuming, subject to deviation with the number of function invocations)

  - In practice, it is usually a trade-off: the best choice or combination have to be found for given application

Source code and associated issues

```
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
```

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

~10% walltime

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

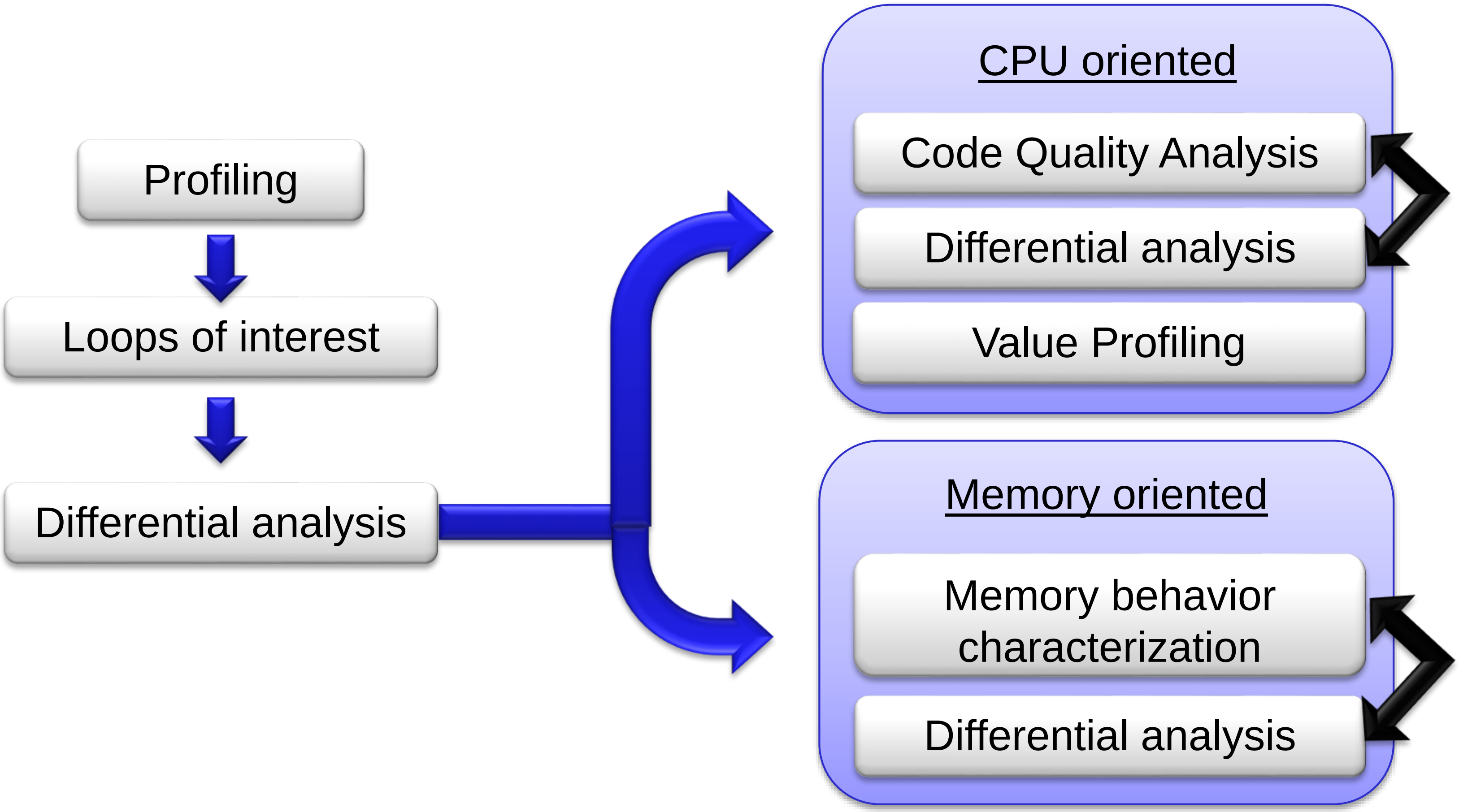
Special issues:
Low trip count: from 2 to 2186 at binary level

Is it possible to:

– **detect all these issues with current tools ?**
– **obtain potential speedup(s) estimation to guide optimization effort ?**

- Our approach: Performance Assessment using MAQAO toolset and Differential Analysis
- Work done at binary level
- Get a global hierarchical view of performance pathologies/bottleneck
- Estimate the performance impact of a given performance pathology while taking into account all of the other pathologies present
- Use different tools for pathology detection and pathology analysis
- Tool selection on pathology basis
- Fine grain - “expensive” - tools only used if necessary on specific issues

- Decision tree:



- Compiler remains our best friend
  - Be sure to select proper flags
    - Know default flags (e.g., -xHost on AVX capable machines)
    - Bypass conservative behavior when possible
  - Pragmas:
    - Vectorization, Alignement, Unrolling, etc…
    - Portable transformations

www.maqao.org

- Open source (LGPL 3.0)
  - Currently binary release
  - Source release soon

- Available for:
  - x86-64
  - Xeon Phi

www.maqao.org

- Audience
  - User/Tool developer:
    - analysis and optimization tool
  - Performance tool developer: framework services
    - BULL SAS: on-going effort – PerfCloud (MIL*)
    - University of Oregon: TAU tool – tau_rewrite (MIL*)
    - ScoreP project: on-going effort – VI-HPS (MIL*)

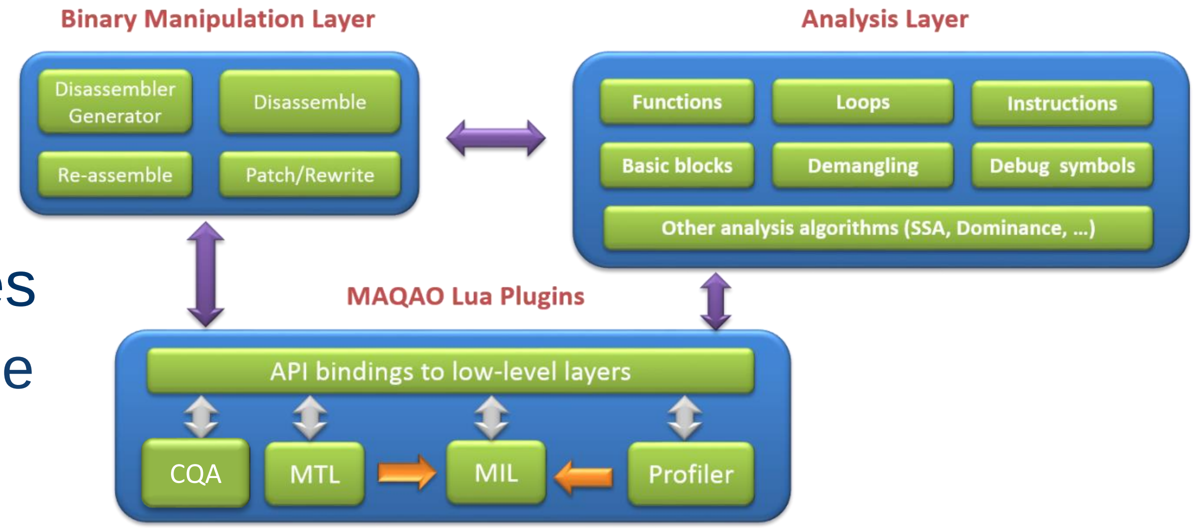
* **M**AQAO **I**nstrumentation **L**anguage

- History
  - Started ten years ago on Itanium
  - Strong emphasis on code generated by the compiler

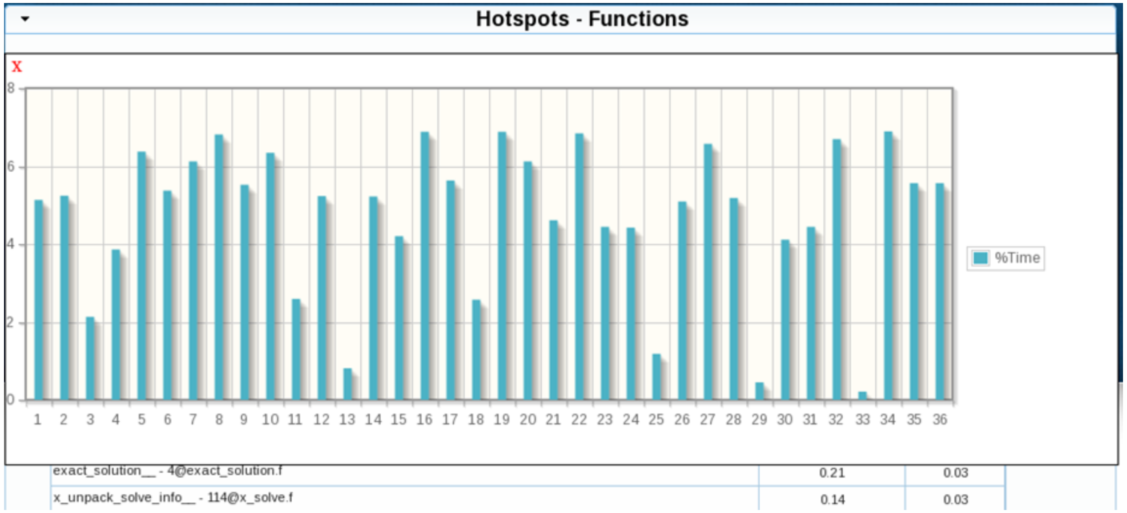
- Contributors
  - ECR (Intel, CEA, GENCI, UVSQ)
  - UVSQ through non-ECR funded projects:
    - H4H
    - PerfCloud
  - University of Bordeaux

MAQAO: Introduction

- Binary level
- Framework services
  - Scripting language
  - Low level API
- Loop-centric (HPC)
- Produce reports
  - We deal with low level details
  - Users get high level reports

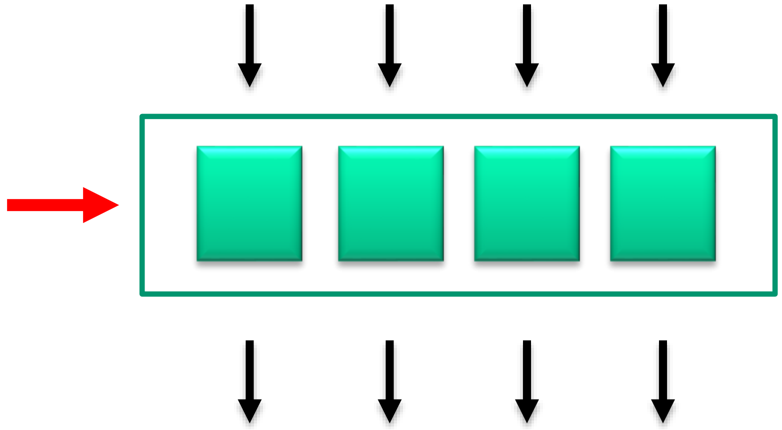


Profiling
Locating hotspots



- Measurement methods
  - Instrumentation
    - Through binary rewriting
    - High overhead / More precision
  - Sampling
    - Hardware counter through perf_event_open system call
    - Very low overhead / less details
  - Default method: Sampling using hardware counters

- Collection level
  - Inter-Node
  - Node
    - Sockets
  - Core level
    - SIMD: data //
    - ILP: instruction level //



- Runtime-agnostic:
  - Only system processes and threads are considered
  - Function hotspots load balancing vue at (multi)node level

- Categorization (MPI/OpenMP/Pthreads/IO/…)

MAQAO PerfEval

- Display functions and their exclusive time
  - Associated callchains and their contribution
  - Loops
- Hardware counters profiles:
  - cache oriented
  - compute oriented
- Innermost loops can then be analyzed by the code quality analyzer module (CQA)
- Command line and GUI (HTML) outputs

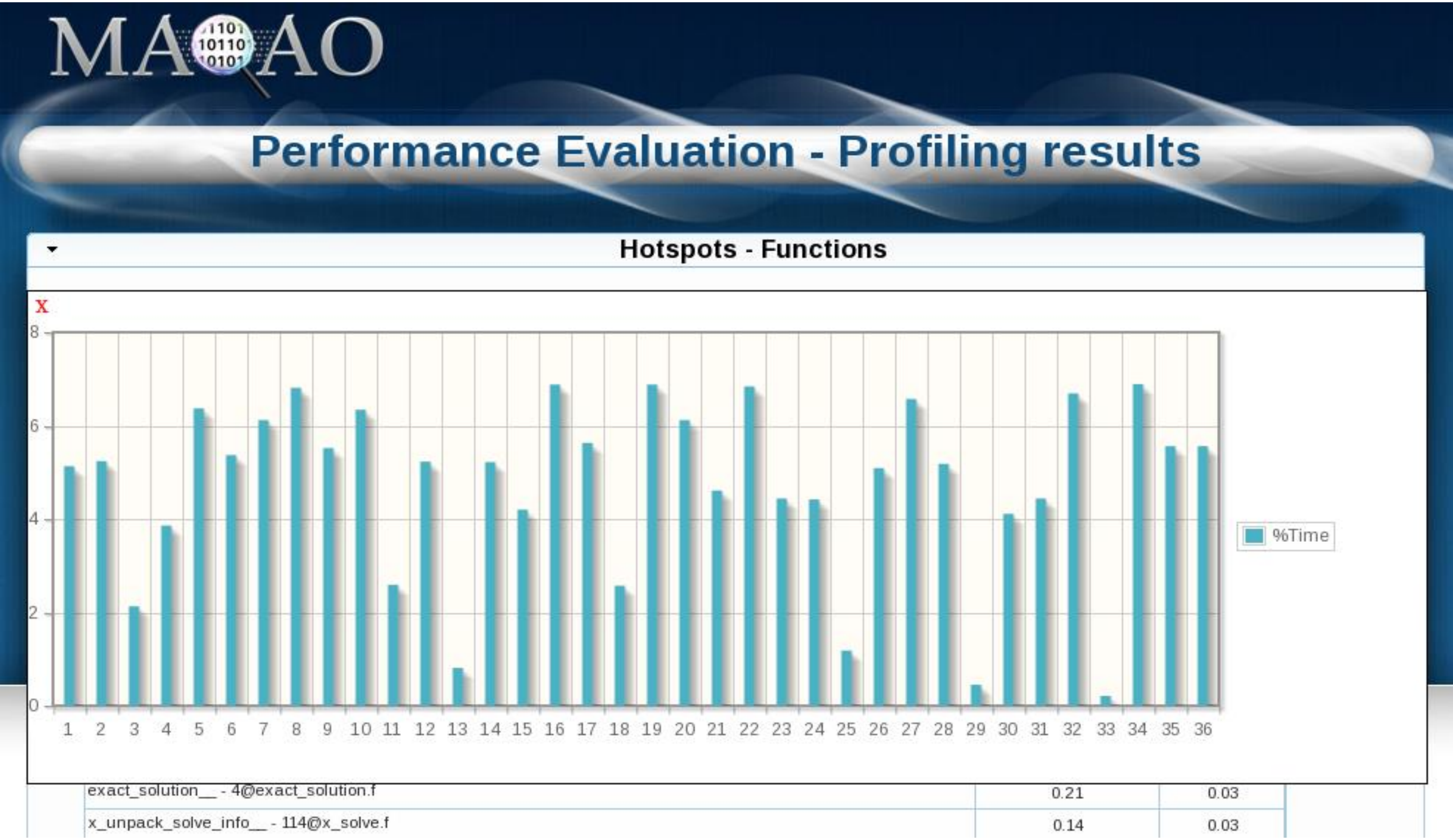
Example: NPB-MPI bt.C 36 processes

Performance Evaluation - Profiling results

Hotspots - Functions

Name	Median Excl %Time	Deviation
matmul_sub__ - 56@solve_subs.f	17.16	0.26
compute_rhs__ - 4@rhs.f	10	0.03
y_solve_cell__ - 385@y_solve.f	9.32	0.54
z_solve_cell__ - 385@z_solve.f	8.96	0.14
x_solve_cell__ - 391@x_solve.f	8.68	0.17
MPIDI_CH3I_Progress	5.22	3.66
matvec_sub__ - 5@solve_subs.f	3.92	0.11
x_backsubstitute__ - 330@x_solve.f	3.09	0.14
y_backsubstitute__ - 329@y_solve.f	2.05	0.03
z_backsubstitute__ - 329@z_solve.f	1.98	0.06
copy_faces__ - 4@copy_faces.f	0.88	0.06
MPID_nem_dapl_rc_poll_dyn_opt_	0.74	0.62
MPID_nem_lmt_shm_start_send	0.68	0.06

(multi)node load balancing vue



Node vue

cirrus5003 - Process #53572 - Thread #1

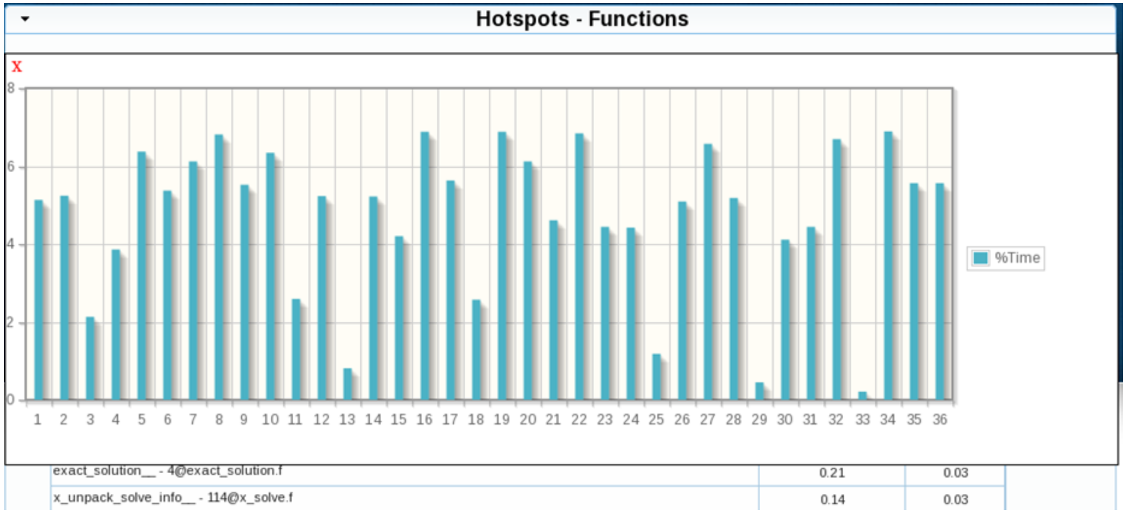
Name	Excl %Time	Excl Time (s)
matmul_sub__ - 56@solve_subs.f	16.92	16.48
compute_rhs__ - 4@rhs.f	9.92	9.66
y_solve_cell__ - 385@y_solve.f	9.08	8.84
loops	9.08	
Loop 267 - y_solve.f@415	0	
Loop 268 - y_solve.f@425	0	
Loop 272 - y_solve.f@426	0.25	
Loop 270 - y_solve.f@524	6.57	
Loop 271 - y_solve.f@436	2.22	
Loop 269 - y_solve.f@716	0.04	
x_solve_cell__ - 391@x_solve.f	9.01	8.78
loops	9.01	
Loop 235 - x_solve.f@420	0	
Loop 236 - x_solve.f@429	0	
Loop 237 - x_solve.f@709	0.06	
Loop 239 - x_solve.f@431	2.71	
Loop 238 - x_solve.f@519	6.24	

Node vue

cirrus5003 - Process #53572 - Thread #1

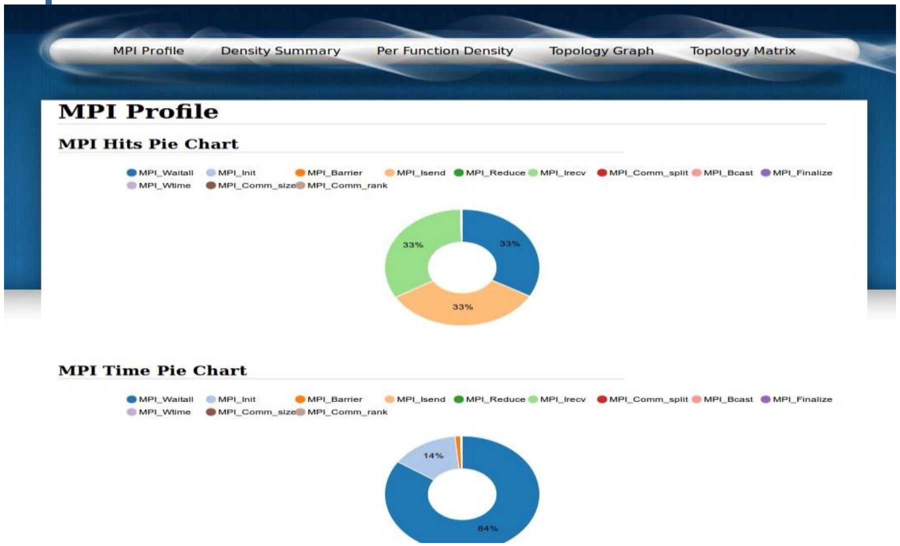
Name	Excl %Time	Excl Time (s)
matmul_sub__ - 56@solve_subs.f	16.92	16.48
compute_rhs__ - 4@rhs.f	9.92	9.66
y_solve_cell__ - 385@y_solve.f	9.08	8.84
loops	9.08	
Loop 267 - y_solve.f@415	0	
Loop 268 - y_solve.f@425	0	
Loop 272 - y_solve.f@426	0.25	
Loop 270 - y_solve.f@524	6.57	
Loop 271 - y_solve.f@436	2.22	
Loop 269 - y_solve.f@716	0.04	
x_solve_cell__ - 391@x_solve.f	9.01	8.78
loops	9.01	
Loop 235 - x_solve.f@420	0	
Loop 236 - x_solve.f@429	0	
Loop 237 - x_solve.f@709	0.06	
Loop 239 - x_solve.f@431	2.71	
Loop 238 - x_solve.f@519	6.24	

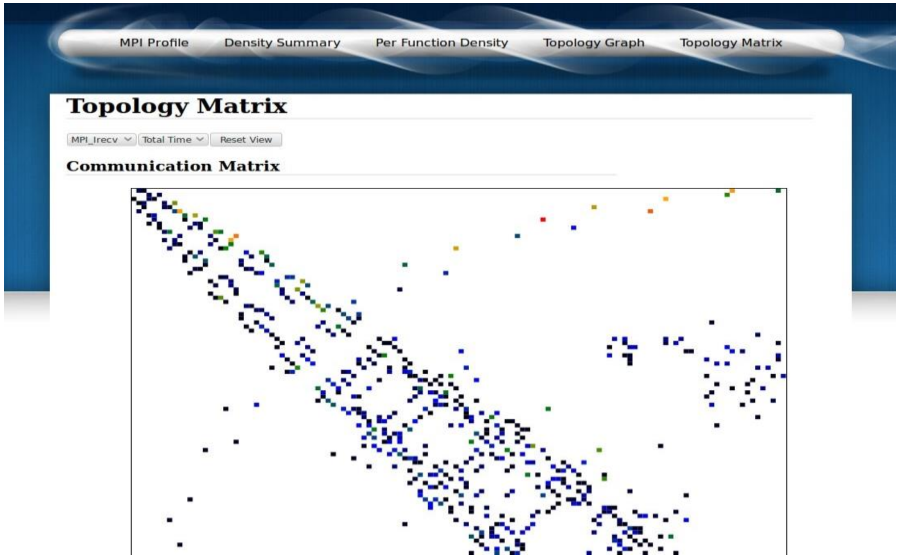
Profiling
Runtime specific tools

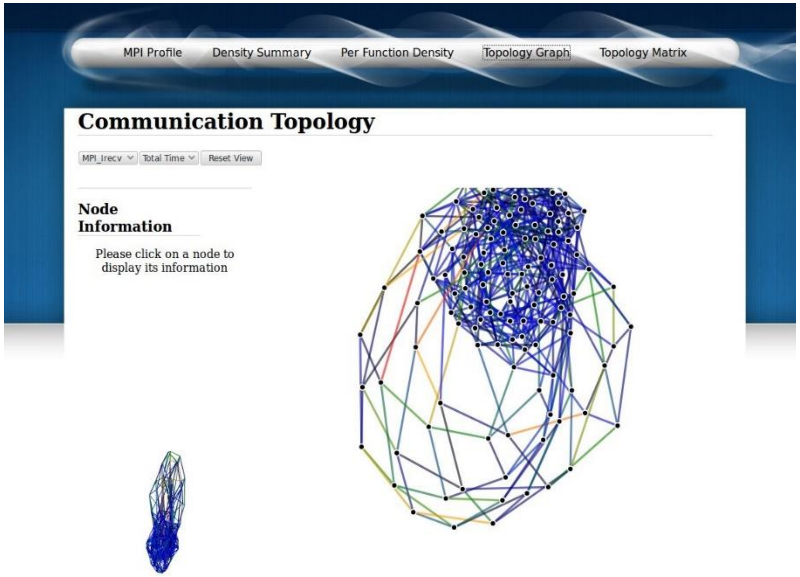


- Online profiling
  - Aggregated metrics (coarse grained analyses)
  - No traces
  - No IOs (only one result file)
  - Reduced memory footprint
  - Scalable on 100+ procs

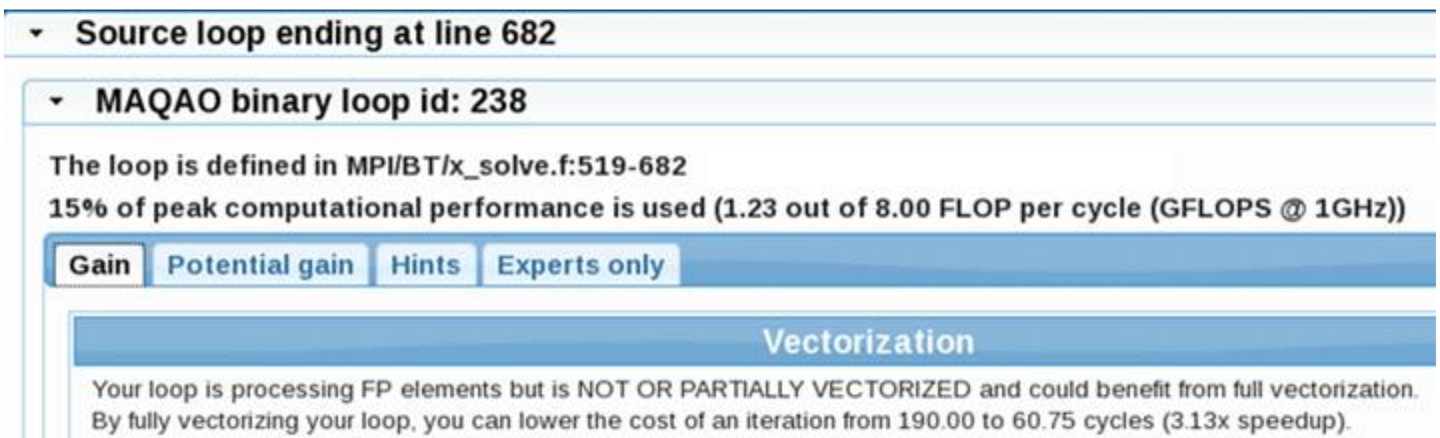
MAQAO PerfEval MPI





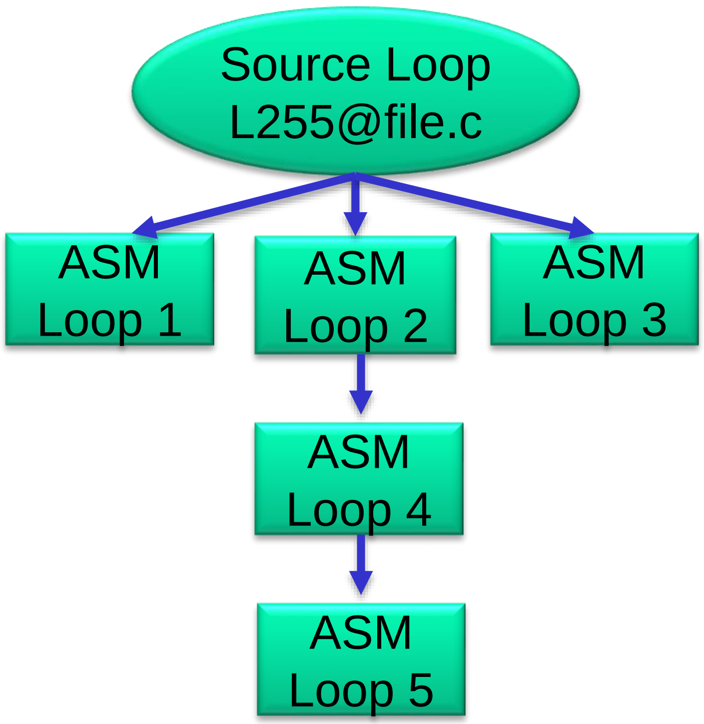


Code Quality Analysis



- Main performance issues:
  - Core level
  - Multicore interactions
  - Communications
- Most of the time core level is forgotten

- Targets innermost loops
  - Source loop versus assembly loop(s)
  - Versioning
  - Peel / Main / Tail
  - Or combination of both



CQA: Code Quality Analyzer

- Simplified static performance model
  - Simulates a target (micro)architecture execution pipeline
  - Instructions description (latency, uops dispatch...)
    - Microbench MAQAO module
  - Out of order considered as ideal
    => no buffers (ROB, RS, PRF)
  - Data is considered resident in L1$
    => Memory issues should be solved before using CQA

CQA: Code Quality Analyzer

- Assess code quality given a binary loop
  - Static performance estimation: lower bounds on cycles
  - Quality metrics:
    - Vectorization degree
    - Impact of address computations (scalar integers)
    - FP contribution (all or pure arith without memory)
    - Detect high latency instructions
    - Unrolling factor detection
  - Provide high level reports
    - Provide source loop context when available
    - Describing a pathology
    - Suggested workarounds to improve static performance
    - Reports categorized by confidence level:
      - gain, potential gain, hint and expert

CQA: Code Quality Analyzer

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

CQA: Code Quality Analyzer

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

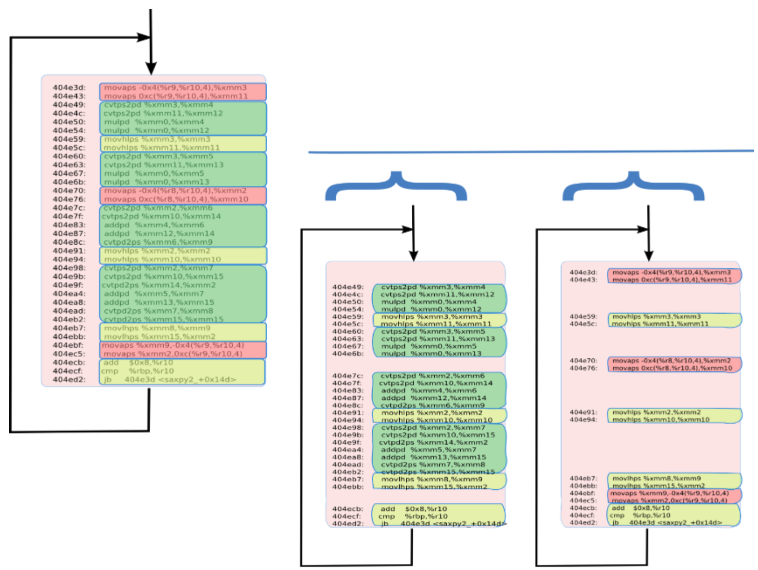
The binary loop is composed of 234 FP arithmetical operations:
- 95: addition or subtraction
- 139: multiply

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.

Differential Analysis



- Targets innermost loops

- Assembly transformations:
  - Insert a new instruction
  - Replace an existing instruction
  - Remove an existing instruction (fill with nops)

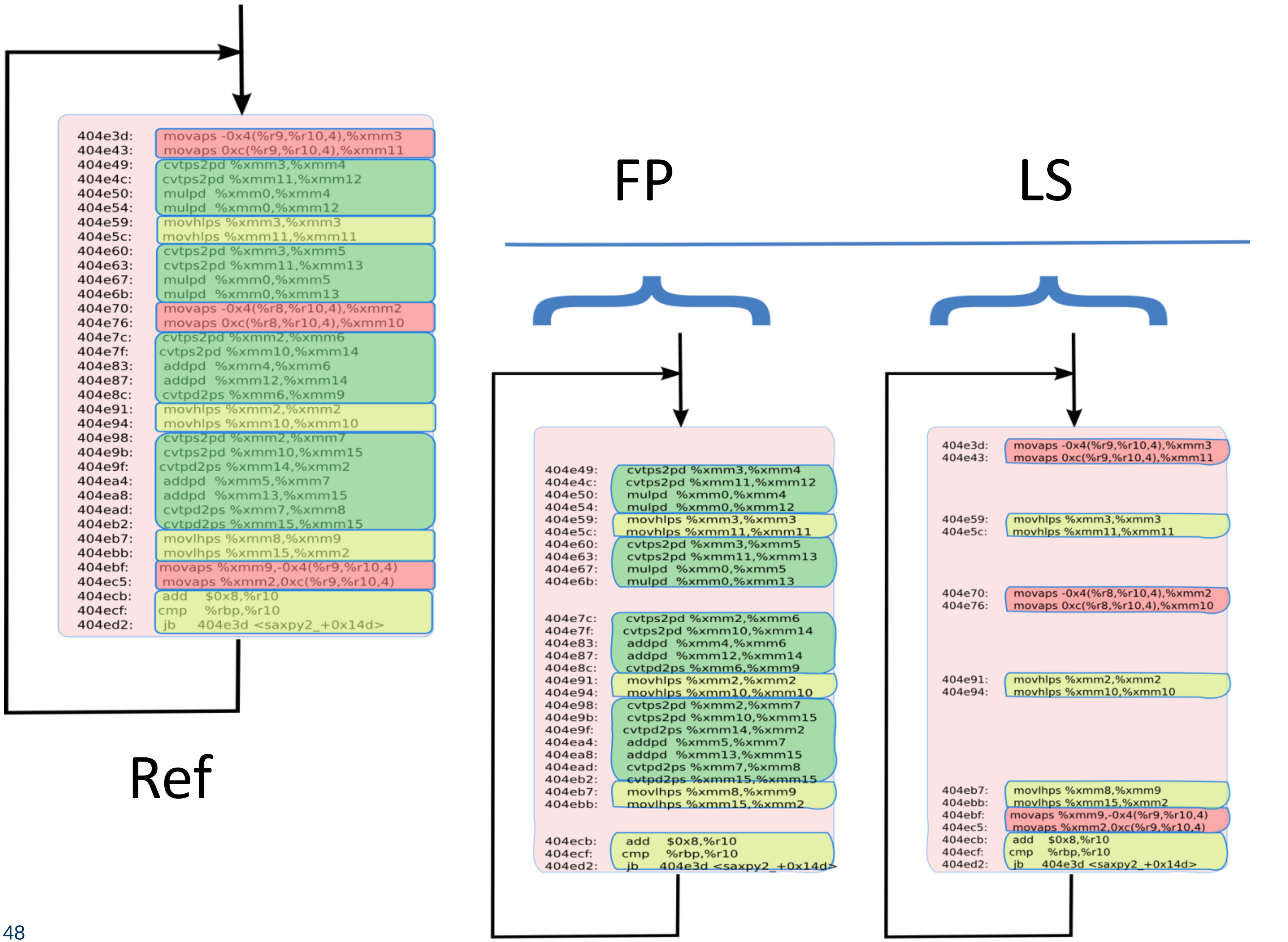
- Differential analysis:
  - Compare the performance of two loops
  - The original binary loop (ref) and a transformed copy of it
  - Goal: create transformations that can
    - Detect bottlenecks
    - Estimate associated ROI

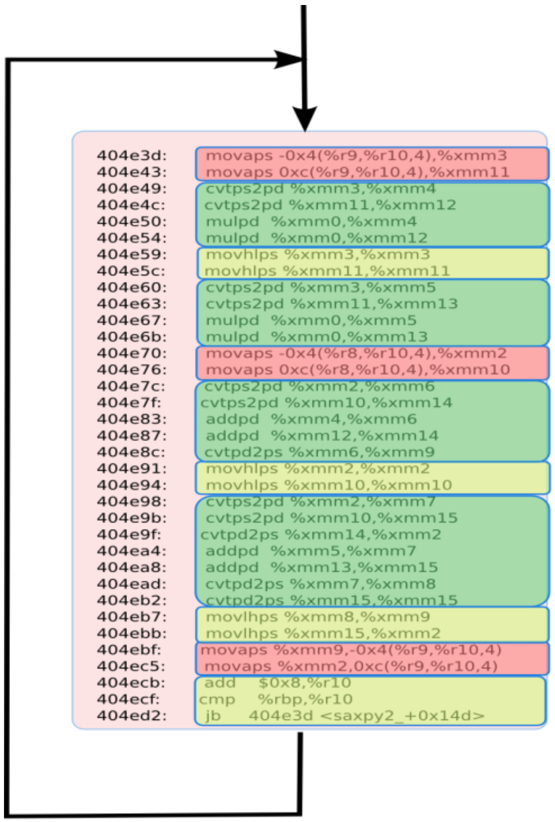
DECAN

- 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

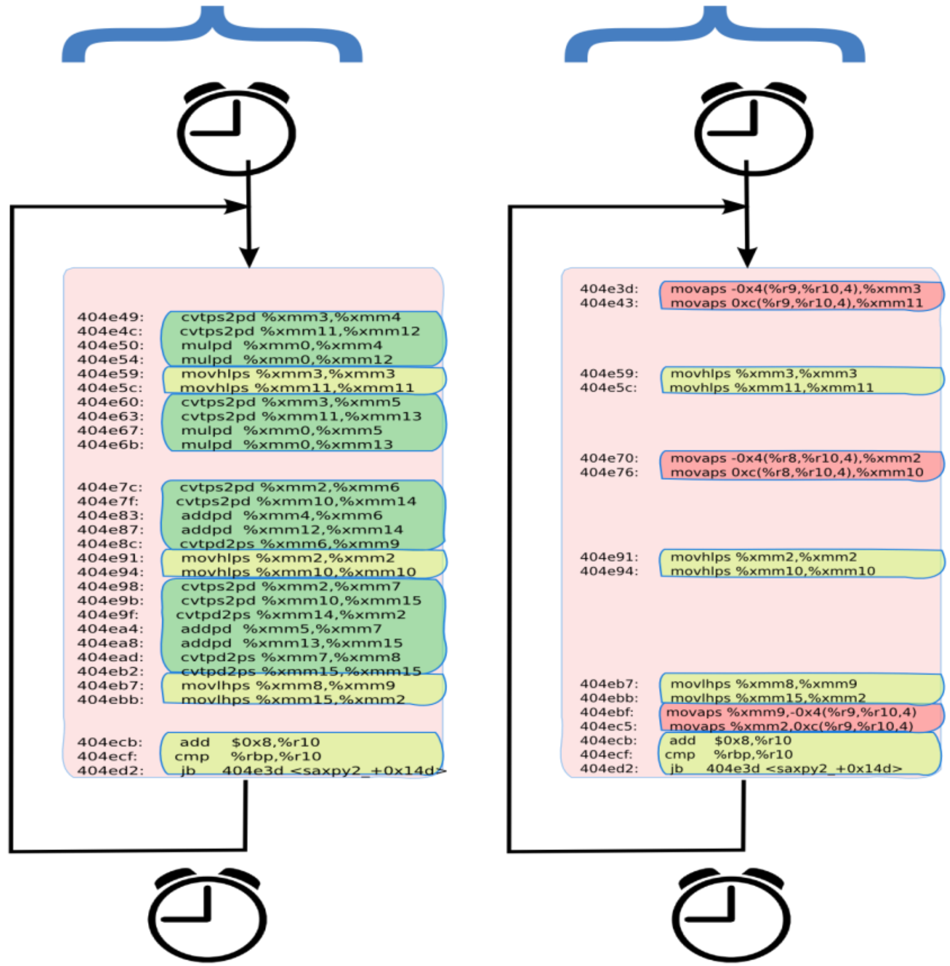
- Typical transformations:
  - FP: only FP arithmetic instructions are preserved
    => loads and stores are removed)
  - LS: only loads and stores are preserved
    => compute instructions are removed)
  - DL1: memory references replaced with global variables ones
    => data now accessed from L1


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>
Ref
FP
LS
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>

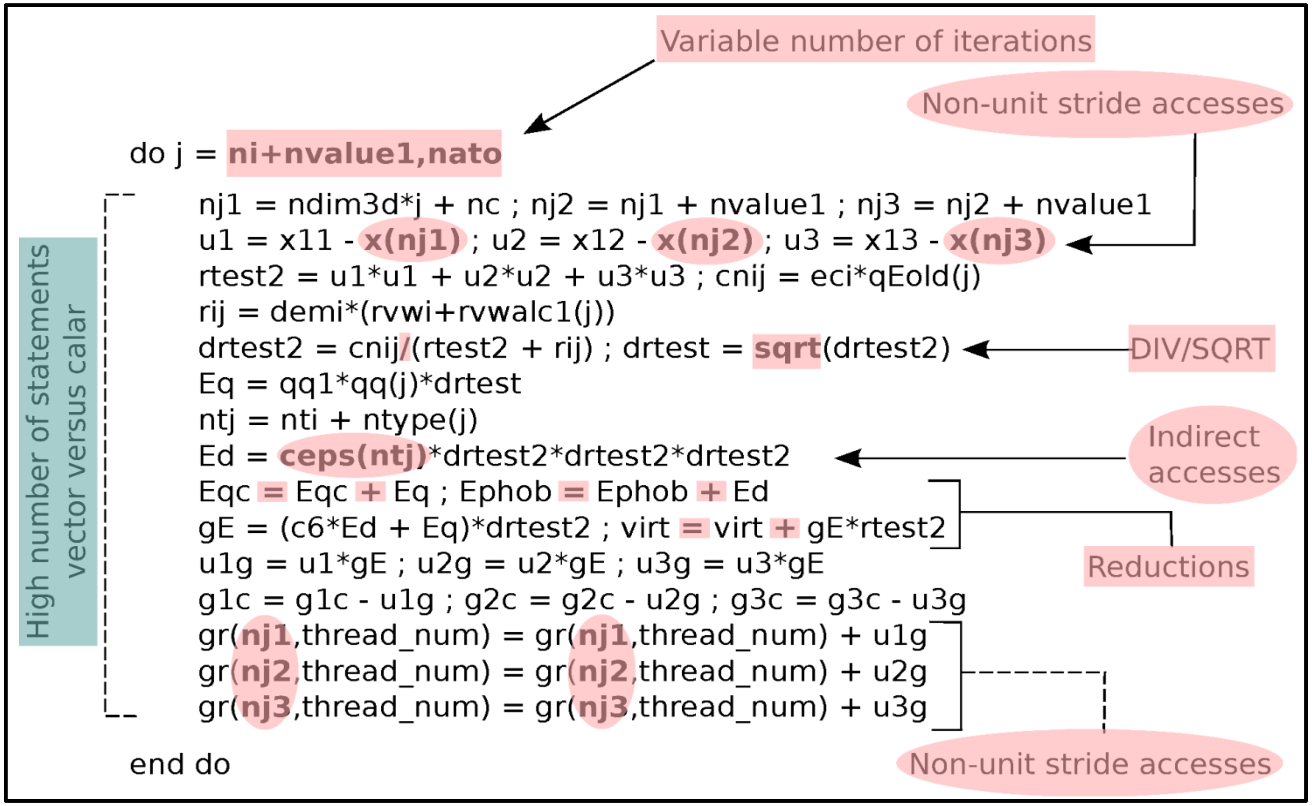


Monitor :

- Execution times
- Loop Iteration numbers
- Hardware counter values



- Polaris: introduction motivating example solution

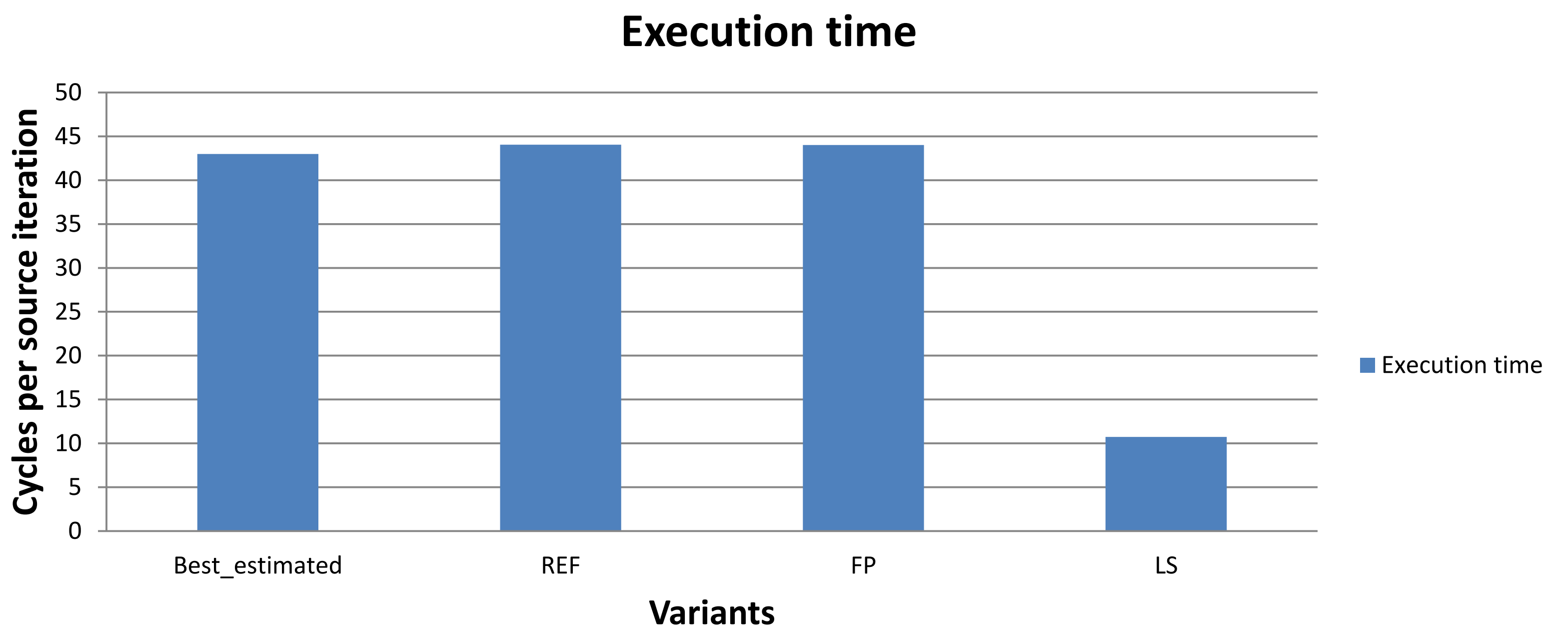


1) High number of statements
2) Non-unit stride accesses
3) Indirect accesses
4) DIV/SQRT
5) Reductions
6) Vector vs Scalar

Special issues:
Low trip count: from 2 to 2186 at binary level

DECAN: Polaris example

- FP / LS transformations

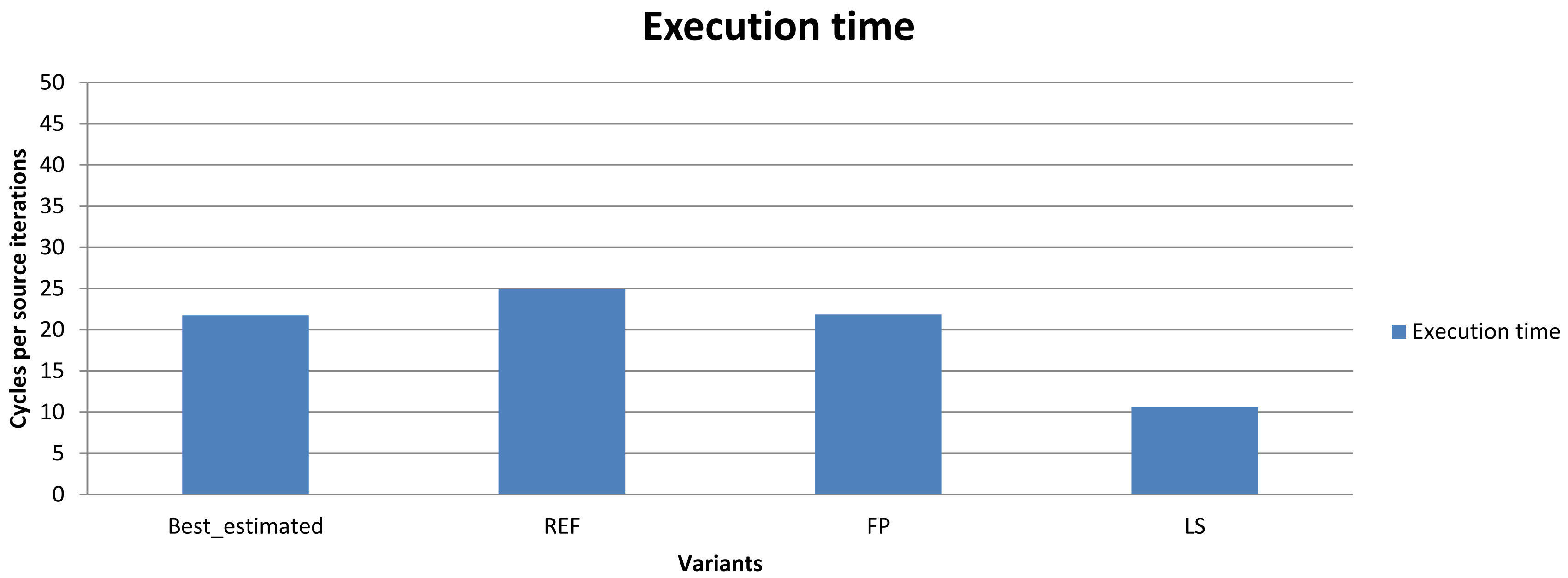


ROI = FP / LS = 4,1
Imbalance between the two streams
=> Try to consume more elements inside one iteration.

DECAN: Polaris example

- FP bound: CQA provides the following metrics:
  - Estimated cycles: 43 (FP = 44)
  - Vector efficiency ratio: 25% (4 DP elements can fit into a 256 bits vector, only 1 is used)
  - DIV/SQRT bound + DP elements:
    - ~4/8x speedup on a 128/256 bits DIV/SQRT unit (2/4x by vectorization + ~2x by using SP)
    - Sandy/Ivy Bridge: still 128 bits (potential speedup 2x DP 4x SP)
    - => First optimization = VECTORIZATION
      - Using SIMD directive
      - Two binary loops
        - Main (packed instructions, 4 elements per iteration)
        - Tail (scalar instructions, 1 element per iteration)

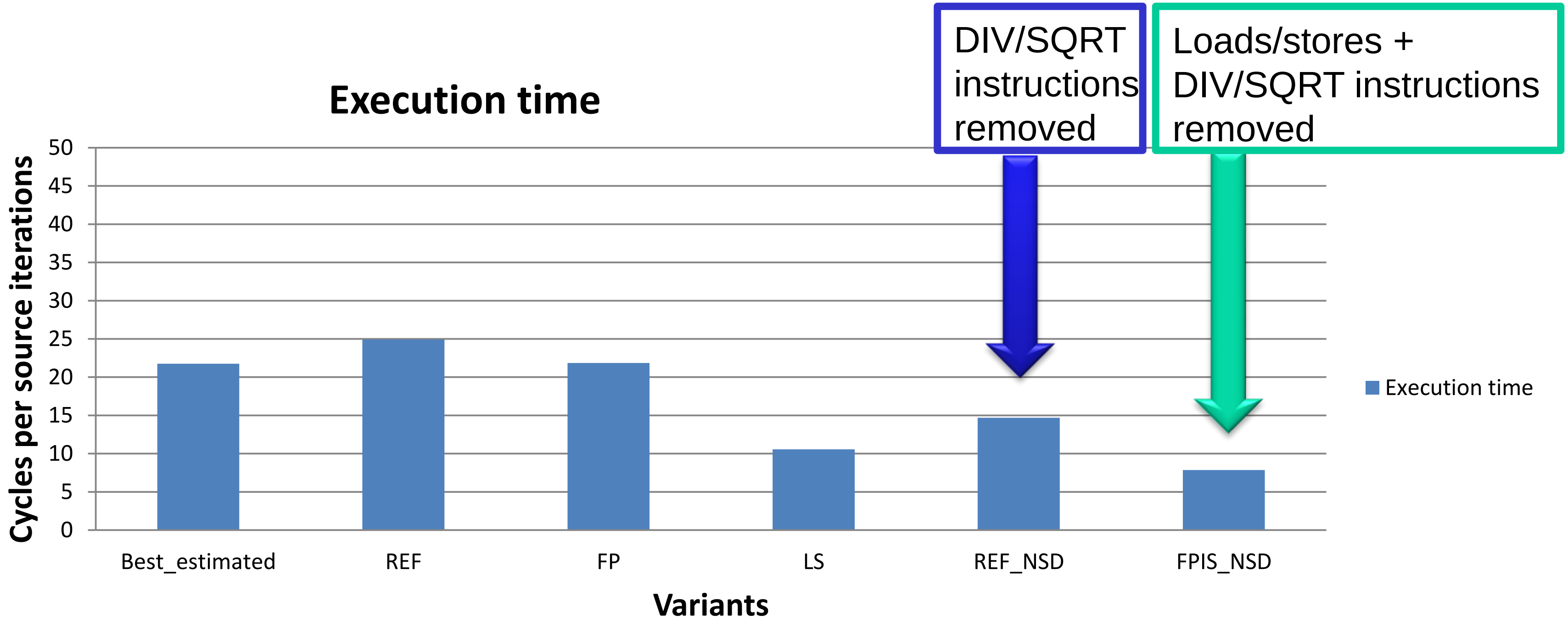
- After vectorization



ROI = FP / LS = 2,07 - Initial ROI was at 4,1

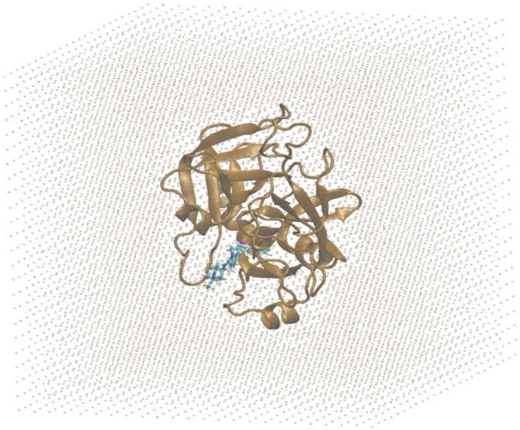
Removing loads/stores provides a speedup much more smaller than removing arithmetical instructions => focus on them

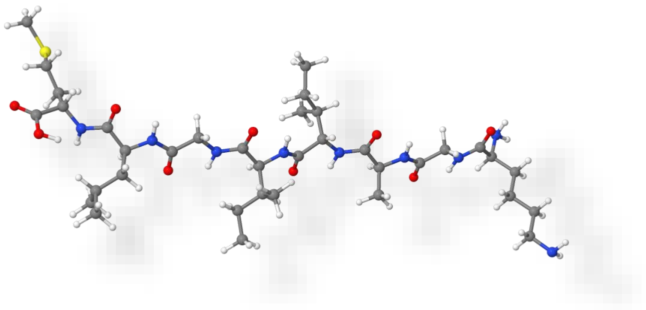
- One step further

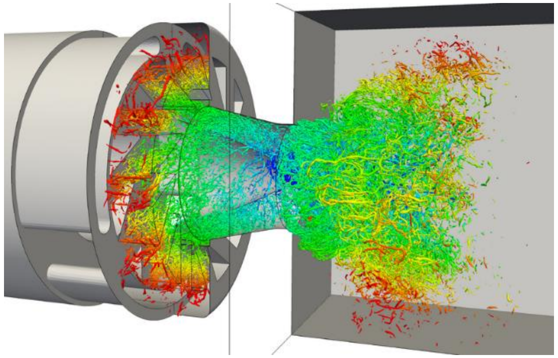


REF_NSD : removing DIV/SQRT instructions provides a 2x speedup
=> the bottleneck is the presence of these DIV/SQRT instructions
FPIS_NSD : removing loads/stores after DIV/SQRT provides a small additional speedup
Conclusion: No room left for improvement here (algorithm bound)

Success stories

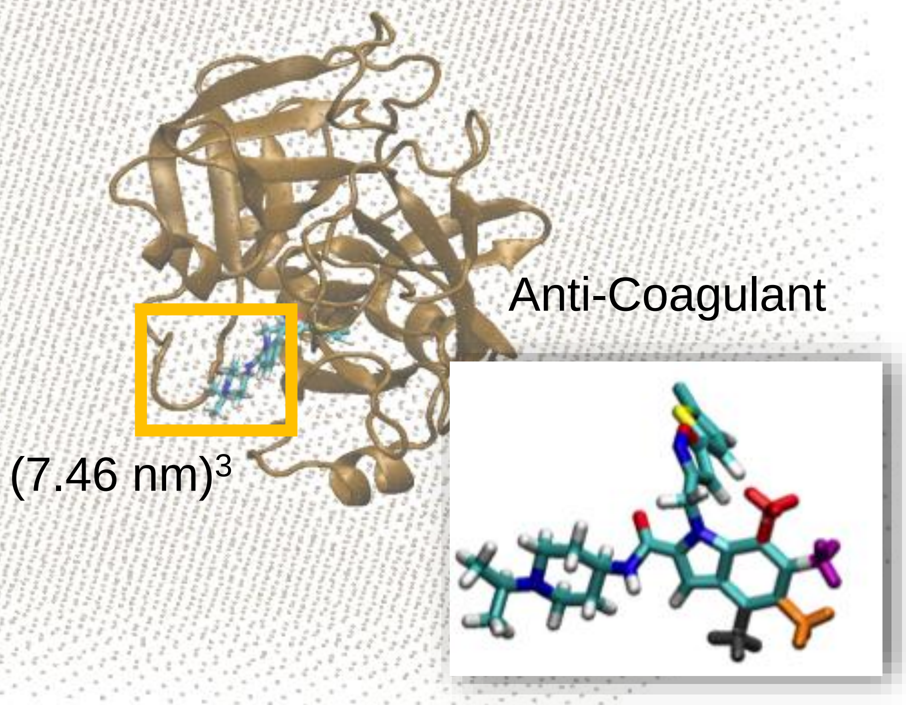






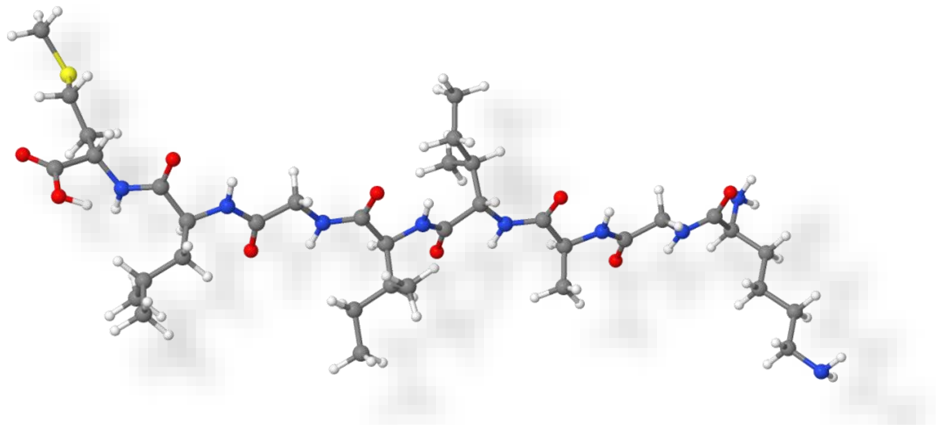
- CEA-DSV : Direction des Sciences du Vivant
- Molecular Dynamics
- **Speedup: 1.5 – 1.7x**
- Effort to speedup:
  - ~ 2 men × months (*)

Example of multi scale problem: Factor Xa, involved in thrombosis



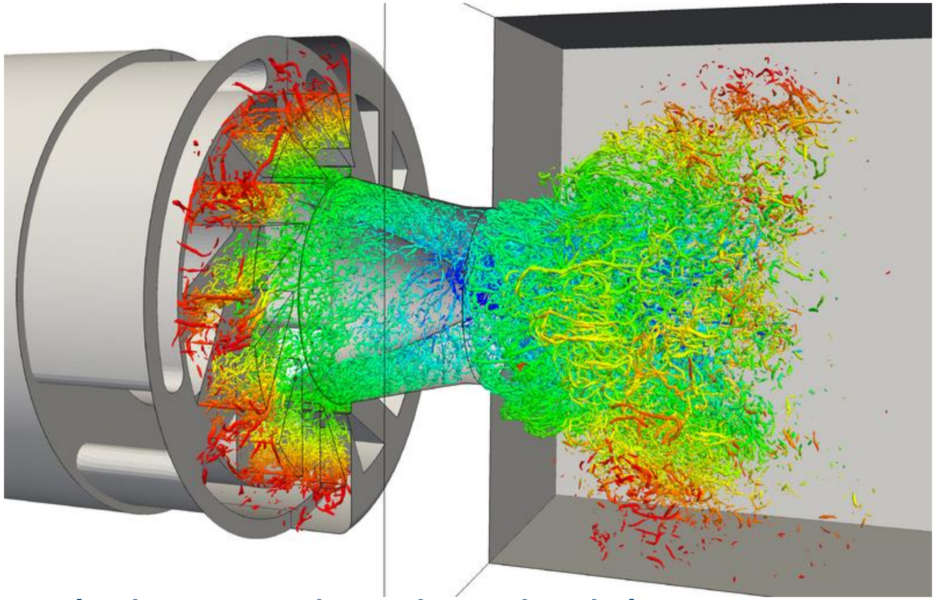
* For the MAQAO team, using ECR tools (MAQAO) and methodology

- IRSAMC : Institut de Recherche sur les Systèmes Atomiques et Moléculaires Complexes
- Quantum chemistry (Monte Carlo)
- **Speedup: > 3x**
- Effort to speedup:
  - ~ 2 men × months (*)



* For the MAQAO team, using ECR tools (MAQAO) and methodology

- CORIA : Complexe de Recherche Inter-professionnel en Aérothermochimie
- Computational fluid dynamics (CFD)
- **Speedup: up to 2.8x**
- Effort to speedup:
  - ~ 3 men × months (*)



* For the MAQAO team, using ECR tools (MAQAO) and methodology

MAQAO

www.maqao.org

Acknowledgements

This work was supported by CEA, GENCI, Intel and UVSQ.

UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

DE LA RECHERCHE À L'INDUSTRIE cea

intel

GENCI

www.maqao.org

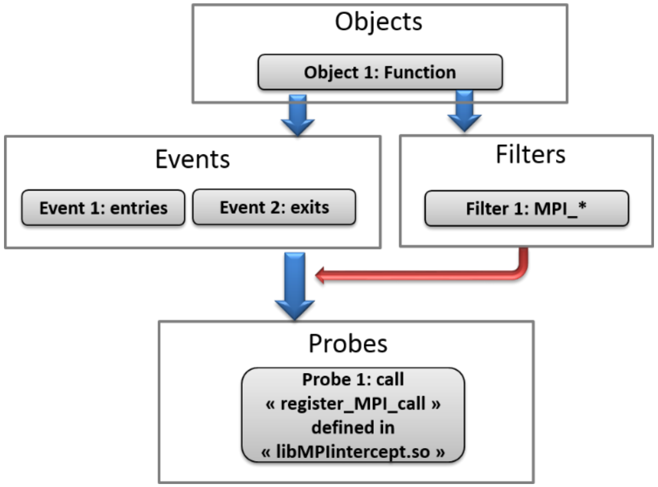
Thanks for your attention !

Questions ?

Meet us @ ECR Booth 24

Backup Slides

MAQAO Instrumentation Language



MIL: MAQAO Instrumentation Language

- A domain specific language to easily build custom 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
- Drive binary manipulation layer of MAQAO tool

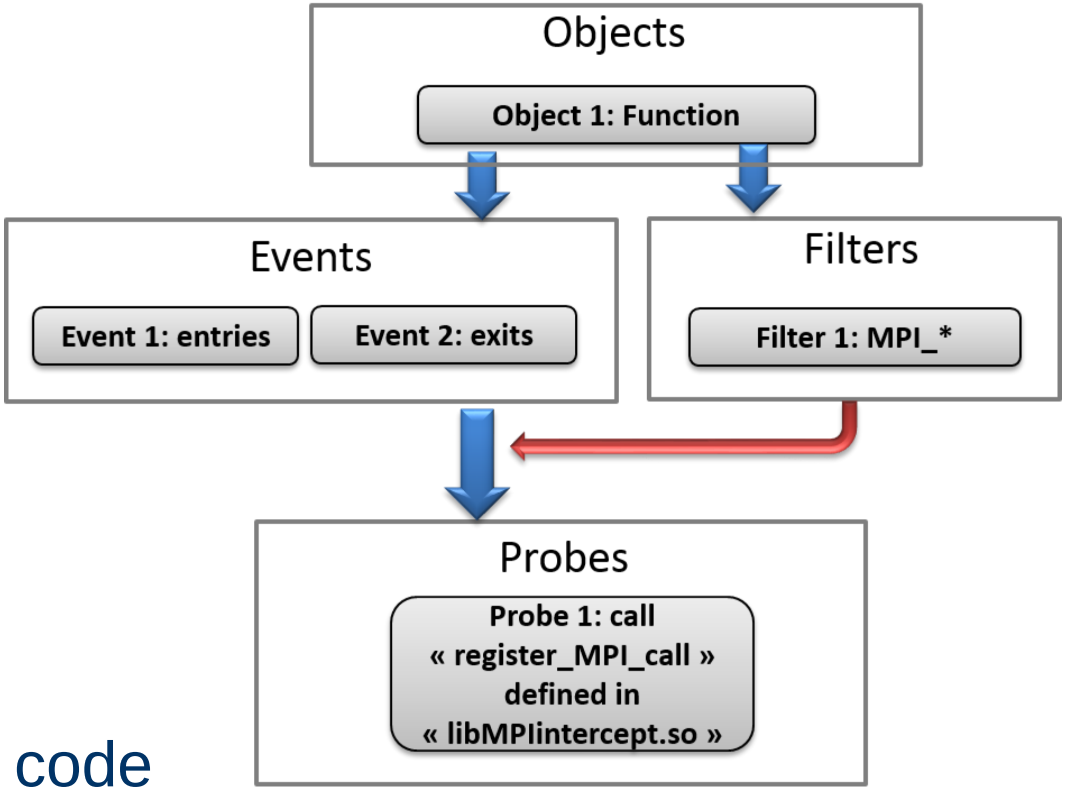
MIL: MAQAO Instrumentation Language

	Dynsinst	PIN	PEBIL
Language type	API Oriented / DSL	API Oriented	API Oriented
Instrumentation type	Static/Dynamic binary	Dynamic binary	Static binary
Overhead	High/High	High	Low
Safe Method	Yes	Yes	No

- Current state of the art:
  - Dyninst appears as the most complete
  - Not sufficient given our goals

MIL: MAQAO Instrumentation Language

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



MIL: MAQAO Instrumentation Language

- Example 1:
  TAU Profiler

```
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);
```

- Object
- Events
- Probes
- Configuration
- Comments

- Example 2: Filtering

```
...
--Instrumentation at function level
fct = mb:addFunction();
--Add some filters (white lists here) using lua regular expressions
fct:addFilterWL('MPI_*');
fct:addFilterWL('GOMP_*');


...
```

- **Object**
- **Filter**
- **Comments**

Previous example only needs an additional statement