


Performance Analysis and Optimization MAQAO Tool

Andrés S. CHARIF-RUBIAL

Emmanuel OSERET

{achar,emmanuel.oseret}@exascale-computing.eu

Exascale Computing Research

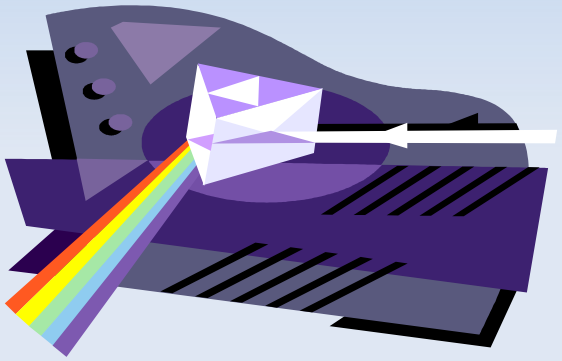
11th VI-HPS Tuning Workshop

Outline

- What's new compared to TW10
  - STAN (ivy bridge), Profiler, Memory characterization
- Introduction
- MAQAO Tool
- Static Analysis
- Dynamic Analysis
- Building performance evaluation tools
- Conclusion

Introduction
Performance analysis

- Understand the performance of an application
  - How well it behaves on a given machine
- What are the issues ?
- Generally a multifaceted problem
  - Maximizing the number of views = better understand
- Use techniques and tools to understand issues
- Once understood ➡ Optimize application



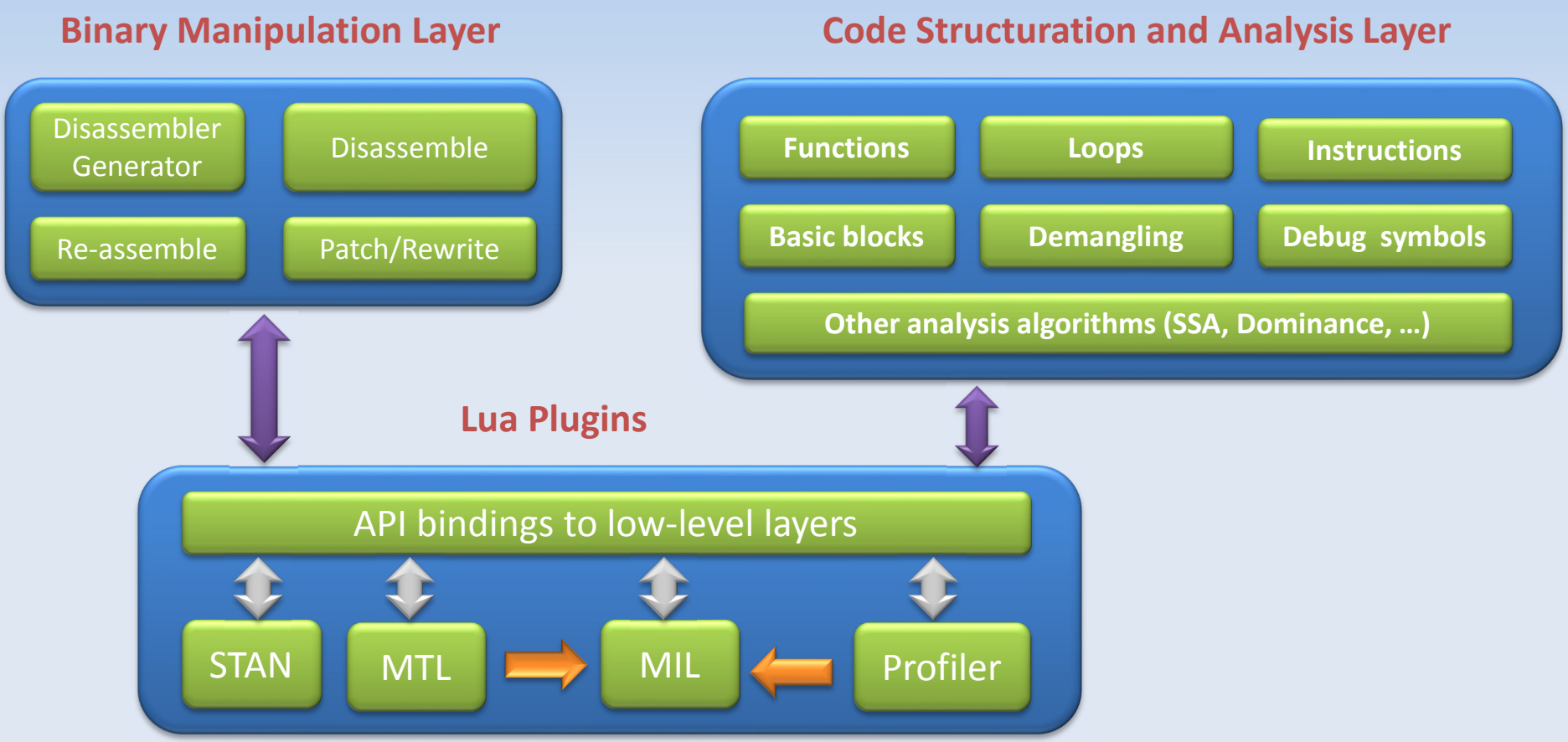
Outline

- Introduction
- MAQAO Tool
- Static Analysis
- Dynamic Analysis
- Building performance evaluation tools
- Conclusion

MAQAO Tool
Components

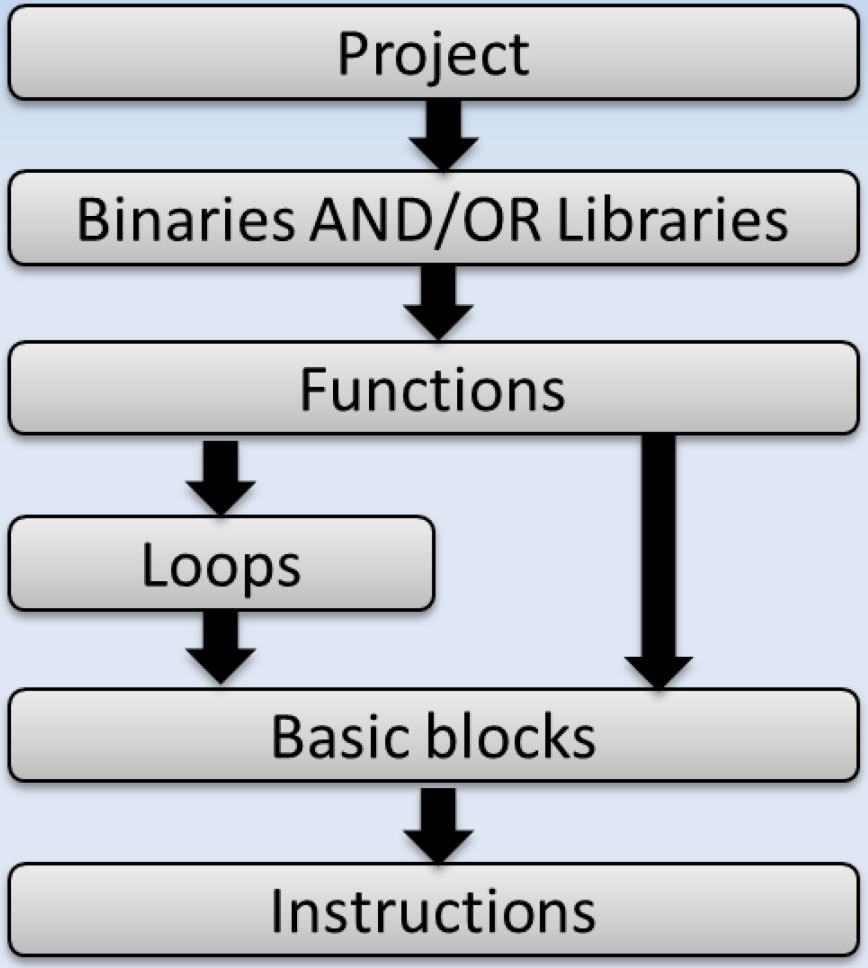
- Framework: a modular approach
  - Expose MAQAO internals
  - Scripting Language (Lua)
  - User defined plugins
- Tool
  - Built on top of the Framework
  - Loop-centric approach
  - Produce reports
  - Methodology: Top/Down iterative

MAQAO Tool
Framework overview



MAQAO Tool
Framework overview

- Hierarchy of objects



MAQAO Tool
Framework overview

- Scripting language
  - Lua language : simplicity and productivity
  - Fast prototyping
  - Lua API : Access to
    - the code structuration and analysis layer
    - the binary rewriting layer
    - already existing modules
  - Customized static analysis
  - Customized dynamic analysis

MAQAO Tool
Framework overview

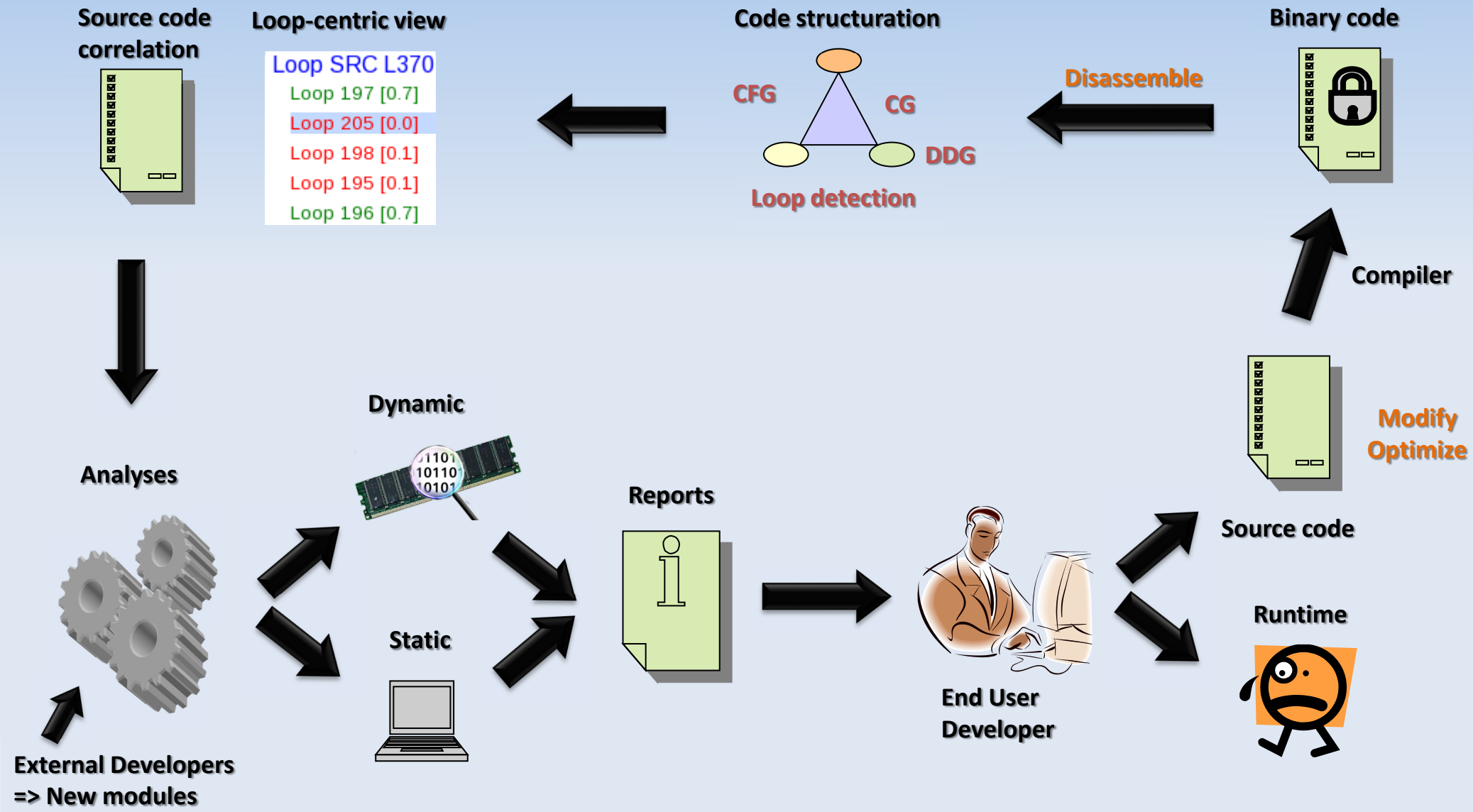
Example of script : Display memory instructions

```
1  --//Create a project and load a given binary
2  local project = project.new ("targeting load memomry instructions");
3  local bin = proj:load ( arg[1], 0);
4  --// Go through the abstract objects hierarchy and filter only load memory instructions
5  for f in bin:functions() do
6    for l in f:innermost_loops() do
7      for b in l:blocks() do
8        for i in b:instructions() do
9          if(i:is_load()) then
10           local memory_operand = i:get_first_mem_oprnd();
11           print(i);
12           print(memory_operand);
13         end
14       end
15     end
16   end
17 end
```

MAQAO Tool
Components

- Framework: a modular approach
  - Expose MAQAO internals
  - Scripting Language (Lua)
  - User defined plugins
- Tool
  - Built on top of the Framework
  - Loop-centric approach
  - Produce reports
  - Methodology: Top/Down iterative

MAQAO Tool
Iterative workflow



MAQAO Tool
Methodology

- Decision tree: smallest possible
- Detect hot spots (functions, loops)
- Specialized modules
  - Memory behavior characterization (MTL)
  - Code quality assessor (STAN)
- Focus on specific problems: value profiling
  - Specialization, Memoization, Instance based VP
  - Ongoing effort

MAQAO Tool
Methodology

- Iterative approach:
  - User chooses to start over again if it is worth
  - Based on reports, modify source code
- Exploit compiler to the maximum
  - Inlining
  - Flags
  - Optimization levels
  - Pragmas : unroll,vectorize
  - Intrinsics
  - Structured code (compiler sensitive)

Outline

- Introduction
- MAQAO Tool
- Static Analysis
- Dynamic Analysis
- Building performance evaluation tools
- Conclusion

Static analysis
STAN module: code quality assesor

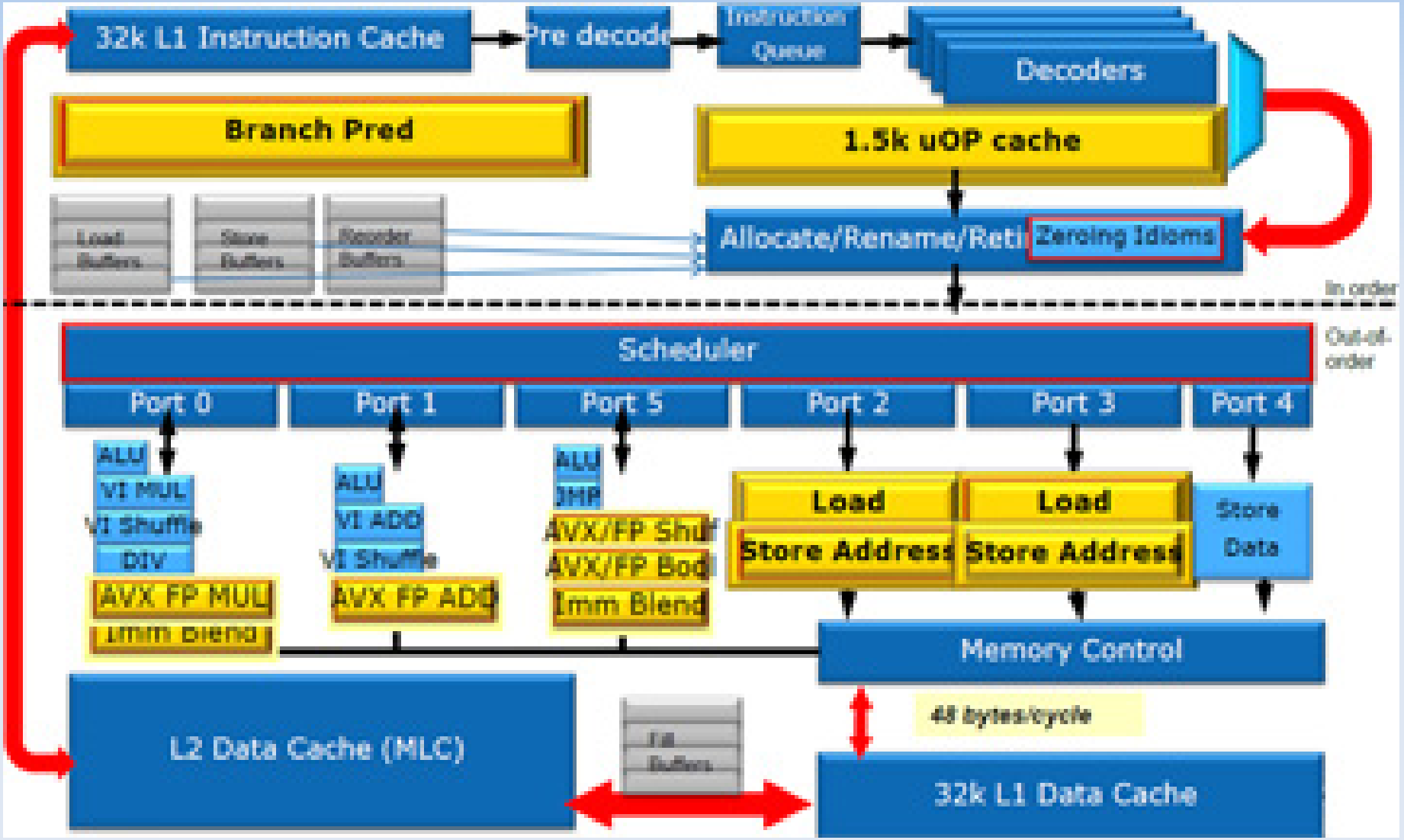
- Loop-centric
- Performance modeling (execution pipeline)
  - Predict performance on one core
- Take into account target (micro)architecture
- Assess code quality
  - Metrics
  - Propose solutions

Static analysis
STAN module: code quality assesor

- Simulates the target micro-architecture
  - Instructions description (latency, uops dispatch...)
  - Machine model

- For a given binary and micro-architecture, provides
  - Quality metrics (how well the binary fits a uarch)
  - Static performance (lower bounds on cycles)
  - Hints and workarounds to improve static performance

Static analysis
Sandy Bridge Pipeline Model



Static analysis
Key metrics

- Unrolling (unroll factor detection)
  - Allows to statically predict performance for different unroll factors
- Vectorization (ratio and speedup)
  - Allows to predict vectorization (if possible) speedup and increase vectorization ratio if it's worth
- High latency instructions (division and square root)
  - Allows to use less precise but faster instructions like RCP (1/x) and RSQRT (1/sqrt(x))

Static analysis

Output example (1/2)

```
Section 1.1.1: Source loop ending at line 7
==========================================

Composition and unrolling
-------------------------
It is composed of the loop 0
and is not unrolled or unrolled with no
peel/tail code (including vectorization).
Type of elements and instruction set
3 SSE or AVX instructions are processing
single precision FP elements in scalar mode
(one at a time).
```

```
Vectorization
-------------
Your loop is not vectorized (all SSE/AVX
instructions are used in scalar mode).

Matching between your loop… and the binary loop
-----------------------------------------------
The binary loop is composed of 1 FP arithmetical
operations:
1: divide
The binary loop is loading 8 bytes (2 single
precision FP elements).
The binary loop is storing 4 bytes (1 single
precision FP elements).

Arithmetic intensity is 0.08 FP operations per
loaded or stored byte.

Cycles and resources usage
--------------------------
Assuming all data fit into the L1 cache, each
iteration of the binary loop takes 14.00 cycles.
At this rate:
 - 0% of peak computational performance is reached
(0.07 out of 16.00 FLOP per cycle (GFLOPS @ 1GHz))
 - 1% of peak load performance is reached (0.57
out of 32.00 bytes loaded per cycle (GB/s @ 1GHz))
 - 1% of peak store performance is reached (0.29
out of 16.00 bytes stored per cycle (GB/s @ 1GHz))
```

Static analysis

Output example (2/2)

Pathological cases

Your loop is processing FP elements but is **NOT OR PARTIALLY VECTORIZED.**
Since your execution units are vector units, only a fully vectorized loop can use their full power.
By fully vectorizing your loop, you can lower the cost of an iteration from 14.00 to 3.50 cycles (4.00x speedup).
Two propositions:
- Try another compiler or update/tune your current one:
  * **gcc: use O3 or Ofast.** If targeting IA32, add mfpmath=sse combined with march=<cputype>, msse or msse2.
  * icc: 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.

WARNING: Fix as many pathological cases as you can before reading the following sections.

Bottlenecks

The divide/square root unit is a bottleneck. Try to reduce the number of division or square root instructions.
If you accept to loose numerical precision, you can speedup your code by passing the following options to your compiler:
gcc: (ffast-math or Ofast) and mrecip
icc: this should be automatically done by default

By removing all these bottlenecks, you can lower the cost of an iteration from 14.00 to 1.50 cycles (9.33x speedup).

Outline

- Introduction
- MAQAO Tool
- Static Analysis
- Dynamic Analysis
- Building performance evaluation tools
- Conclusion

Dynamic analysis

- Static analysis is optimistic
  - Data in L1$
  - Believe architecture
- Get a real image:
  - Coarse grain : find hotspots
  - Fine grain: specialized modules
    - Memory
    - Code quality
  - MIL : specialized instrumentation

Dynamic analysis
Detecting hotspots

- Most time consuming functions and loops
  - Sampling method with HPCs: low overhead
  - Handles serial and OpenMP applications (next target = MPI).
  - Exlusive % time

Dynamic analysis
Detecting hotspots

- Functions

Thread #10762 – OMP #0

Function Name	Time %
compute_rhs_#omp#region#1	25.35
binvcrhs_	22.73
matmul_sub_	11.27
y_solve_#omp#loop#1	10.65
z_solve_#omp#loop#1	9.82
x_solve_#omp#loop#1	8.58
__kmp_wait_sleep [libiomp5.so]	3.89
matvec_sub_	2.82
add_#omp#loop#1	2.19
__kmp_x86_pause [libiomp5.so]	1.26
Others	0.71
lhsinit_	0.21
__kmp_yield [libiomp5.so]	0.18
binvrhs_	0.12
exact_solution_	0.12
exact_rhs_#omp#region#1	0.05
initialize_#omp#region#1	0.02
__kmpc_barrier [libiomp5.so]	0.01
__kmpc_for_static_init_4 [libiomp5.so]	0.01
__kmp_launch_thread [libiomp5.so]	0.01
__kmp_run_before_invoked_task [libiomp5.so]	0.00
data.10538 [libc-2.12.so]	0.00
kmp_threadprivate_insert [libiomp5.so]	0.00
__kmpc_for_static_fini [libiomp5.so]	0.00
__kmp_task_team_sync [libiomp5.so]	0.00
__kmp_invoke_microtask [libiomp5.so]	0.00

Dynamic analysis
Detecting hotspots

- Loops

Thread #10762 – OMP #0

Loop ID	Source Infos	Level	Time %
111	compute_rhs_#omp#region#1 - 291,336	Innermost	6.12
181	y_solve_#omp#loop#1 - 136,298	Innermost	4.33
191	z_solve_#omp#loop#1 - 137,299	Innermost	4.35
171	x_solve_#omp#loop#1 - 137,299	Innermost	4.07
182	y_solve_#omp#loop#1 - 46,128	Innermost	3.77
192	z_solve_#omp#loop#1 - 46,128	Innermost	3.23
96	compute_rhs_#omp#region#1 - 375,379	Innermost	3.24
149	compute_rhs_#omp#region#1 - 70,119	Innermost	3.08
133	compute_rhs_#omp#region#1 - 181,225	Innermost	2.95
161	compute_rhs_#omp#region#1 - 26,36	Innermost	2.90
156	compute_rhs_#omp#region#1 - 52,53	Innermost	2.48
172	x_solve_#omp#loop#1 - 48,130	Innermost	2.51
198	add_#omp#loop#1 - 22,23	Innermost	2.17
176	y_solve_#omp#loop#1 - 386,389	InBetween	1.14
141	compute_rhs_#omp#region#1 - 144,148	Innermost	1.19
186	z_solve_#omp#loop#1 - 399,402	InBetween	1.20
166	x_solve_#omp#loop#1 - 387,390	InBetween	0.99
79	compute_rhs_#omp#region#1 - 419,420	Innermost	0.83
121	compute_rhs_#omp#region#1 - 252,256	Innermost	0.78
189	z_solve_#omp#loop#1 - 399,402	Innermost	0.44
179	y_solve_#omp#loop#1 - 386,389	Innermost	0.34
165	x_solve_#omp#loop#1 - 387,390	Innermost	0.38
112	compute_rhs_#omp#region#1 - 291,336	Innermost	0.41
185	z_solve_#omp#loop#1 - 399,402	Innermost	0.28
169	x_solve_#omp#loop#1 - 387,390	Innermost	0.38
175	y_solve_#omp#loop#1 - 386,389	Innermost	0.30
188	z_solve_#omp#loop#1 - 399,402	Innermost	0.19
168	x_solve_#omp#loop#1 - 387,390	Innermost	0.21
134	compute_rhs_#omp#region#1 - 181,225	Innermost	0.20
150	compute_rhs_#omp#region#1 - 70,119	Innermost	0.15
170	x_solve_#omp#loop#1 - 333,353	Innermost	0.17

. . .

Dynamic analysis
Memory behavior characterisation

- Transformation opportunities, e.g.: loop interchange

- Data reshaping opportunities , e.g.: array splitting

Memory behavior characterization

Single threaded aspects: Inefficient patterns

Real code example: PMBENCH

```
for (int n=0; n<M; n++)
  if (lambdaz[n] > 0.) {
    for (int j=0; j<mesh.NCx; j++)
      for (int i=1; i<mesh.NCz; i++)
        J_upz[IDX3C(n,i,M,j,(mesh.NCz+1)*M)] = Jz[IDX3C(n,i-1,M,j,(mesh.NCz)*M)] * lambdaz[n];
  }
  if (lambdaz[n] < 0.){
```

MTL output

Load (Double) - Pattern: **8*i1** (Hits : 100% | Count : 1)
Load (Double) - Pattern: **8*i1+217600*i2+1088*i3** (Hits : 100% | Count : 1)
Store (Double)- Pattern: **8*i1+218688*i2+1088*i3** (Hits : 100% | Count : 1)

- Stride 1 (8/8) one access for outmost
- Poor access patterns for two instructions
- Idealy: smallest stides inside to outside
- Here: interchange **n** and **i** loops

Memory behavior characterization

Single threaded aspects: Inefficient patterns

Real code example: PMBENCH

- Example: **flux_numerique_z,** loop **193** (same for 195)
- Same kind of optimization for loops 204 and 206

Original

```
for (int n=0; n<M; n++) {
  if (lambdaz[n] > 0.){
    for (int j=0; j<NCx; j++)
      for (int i=1; i<NCz; i++) // loop 193
       J_upz[IDX3C(n,i,M,j,(NCz+1)*M)]=
Jz[IDX3C(n,i-1,M,j,(NCz)*M)] * lambdaz[n];
  }
  if (lambdaz[n] < 0.)
    …//loop 195
}
```

After transformation

```
for (int j=0; j<NCx; j++)
  for (int n=0; n<M; n++) {
    if (lambdax[n] > 0.){
      for (int i=1; i<NCz; i++) // loop 193
       J_upz[IDX3C(n,i,M,j,(NCz+1)*M)]=
Jz[IDX3C(n,i-1,M,j,(NCz)*M)] * lambdaz[n];
  }
  if (lambdaz[n] < 0.)
    …//loop 195
  }
```

7.7x local speedup (loops) ⟹ 1.4x GLOBAL speedup

Outline

- Introduction
- MAQAO Tool
- Static Analysis
- Dynamic Analysis
- Building performance evaluation tools
- Conclusion

MIL: Instrumentation language
New DSL

- A domain specific language to easily build 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 (rewriting)
  - Efficient: lowest overhead
  - Robust: ensure the program semantics
  - Accurate: correctly identify program structure
- Drive binary manipulation layer

Instrumentation language
Language concepts/features

- Global variables (in binary)
- Events
- Filters
- Actions
- Configuration features
  - Output
  - Language behavior (properties)
- Runtime embedded code

Instrumentation Language
Language concepts/features

- Events: Where ?

Level	Events
Program	Entry / Exit (avoid LD + exit handlers)
Function	Entries / Exits
Loop	Entries / Exits / Backedges
Block	Entry / Exit
Instruction	Before / After
Callsite	Before / After

Instrumentation Language
Language concepts/features

- Probes: What ?
  - External functions
    - Name
    - Library

```
name = "traceEntry",
lib = "libTauHooks.so",
params = { {type = "macro",value = "profiler_id"} }
```

    - Parameters: int,string,macros,function (static⇔dynamic)
    - Return value
    - Demangling

```
_ZN3MPI4CommC2Ev
MPI::Comm::Comm()
```

    - Context saving
  - ASM inline: gcc-like
  - Runtime embedded code (lua code within MIL file)

Instrumentation Language
Language concepts/features

- Filters:
  - Why ? Reduce instrumentation probes
    - Target what really matters
  - Lists: regular expressions
    - White list
    - Black list
  - Built-in: structural properties attributes
    - Example: nesting level for a loop
  - User defined: an action that returns true/false

Instrumentation Language
Language concepts/features

- Actions:
  - Why ? For complex instrumentation queries
  - Scripting ability (Lua code)
  - User-defined functions
  - Access to MAQAO Plugins API (existing modules)

Instrumentation Language
Language concepts/features

- Passes:
  - To address complex multistep instrumentations
  - Example: detect OpenMP events
    - Step 1: static analysis to detect sequences of call sites
      - Only events and actions are used
    - Step 2: instrument
      - Select (same or new) events and insert probes based on step 1

Conclusion

- Select a consistent methodology
- Detect hotspots
- Use specialized modules at loop level
  - Assess code quality through static analysis
  - Memory behavior characterization
- If no relevant existing module : use MIL
- Iterative approach

Thanks for your attention !

Questions ?

Hand-on exercises

- Refer to the exercises folder
  - cd $UNITE_ROOT/tutorial/maqao/
  - Read Outline.txt