Performance Analysis and Optimization MAQAO Tool

Andrés S. CHARIF-RUBIAL
Emmanuel OSERET
{achar,emmanuel.oseret}@exascale-computing.eu
Exascale Computing Research

VI-HPS Workshop

Outline

- Introduction
- MAQAO Tool and Framework
- Static Analysis
- Building performance evaluation tools
- Conclusion

Methodology

- Type of code ? CPU or memory bound
- Static + Dynamic approach
- Approach : Top-Down / Iterative
- Detect hot spots
- Focus on specific parts

Methodology

- Exploit compiler to the maximum
  - IPO and inlining !!!
  - Flags
  - Optimization levels
  - Pragmas : unroll,vectorize
  - Intrinsics
  - Structured code (compiler sensitive)

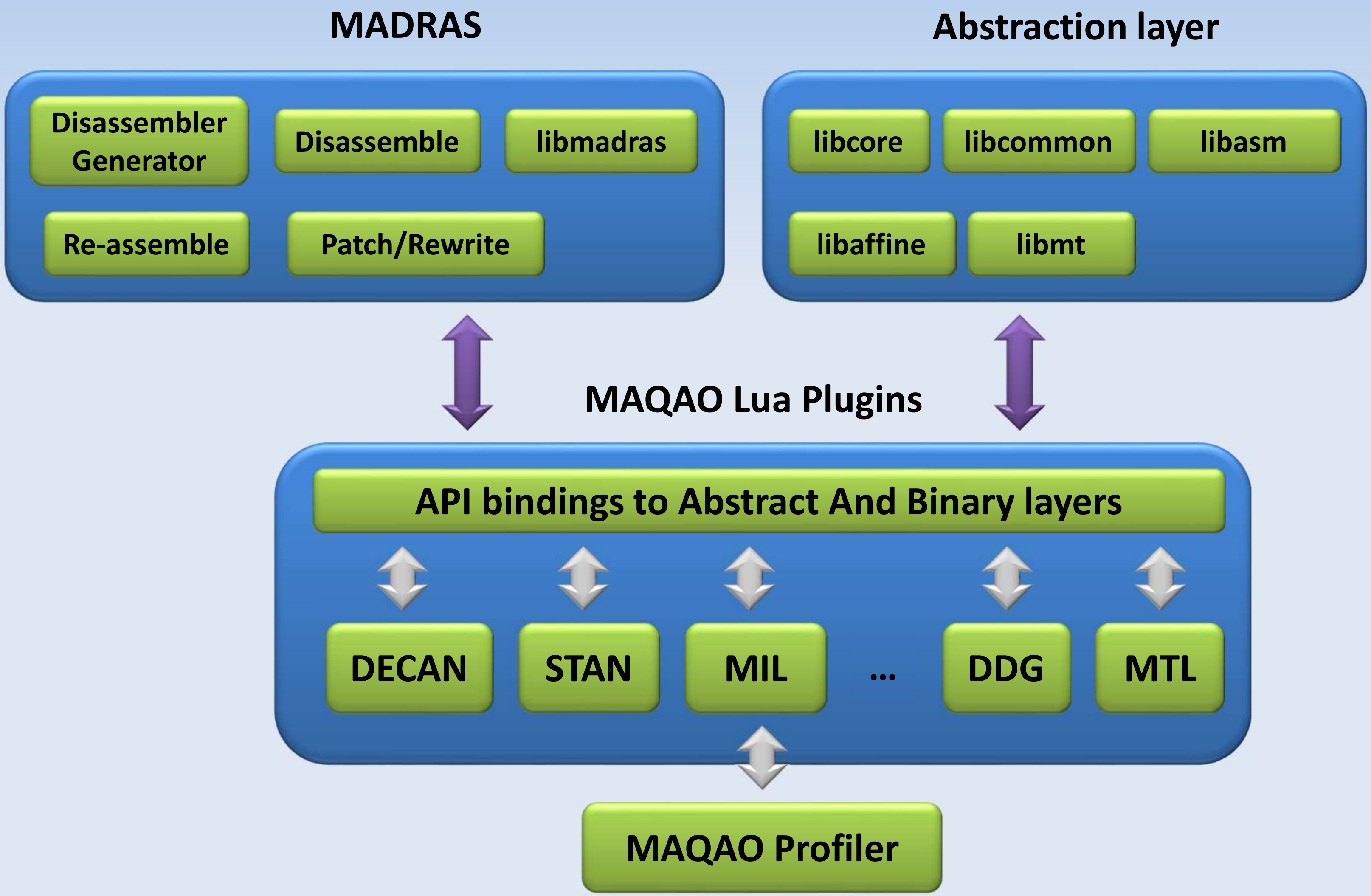
MAQAO Tool and Framework

- MAQAO Framework
  - Modular approach
  - Reusable components
- MAQAO Tool
  - Using Framework
  - Scripting Language
  - Batch interface

MAQAO Framework

- Binary manipulation
- Set of C libraries (core features)
- Scripting language on top
- Plugins

MAQAO Framework



MAQAO Framework

- Scripting language
  - Lua language : simplicity and productivity
  - Fast prototyping
  - MAQAO Lua API : Access to
    - an abstraction layer
    - a binary rewriting layer
    - already existing modules
  - Customized static analysis
  - Customized dynamic analysis

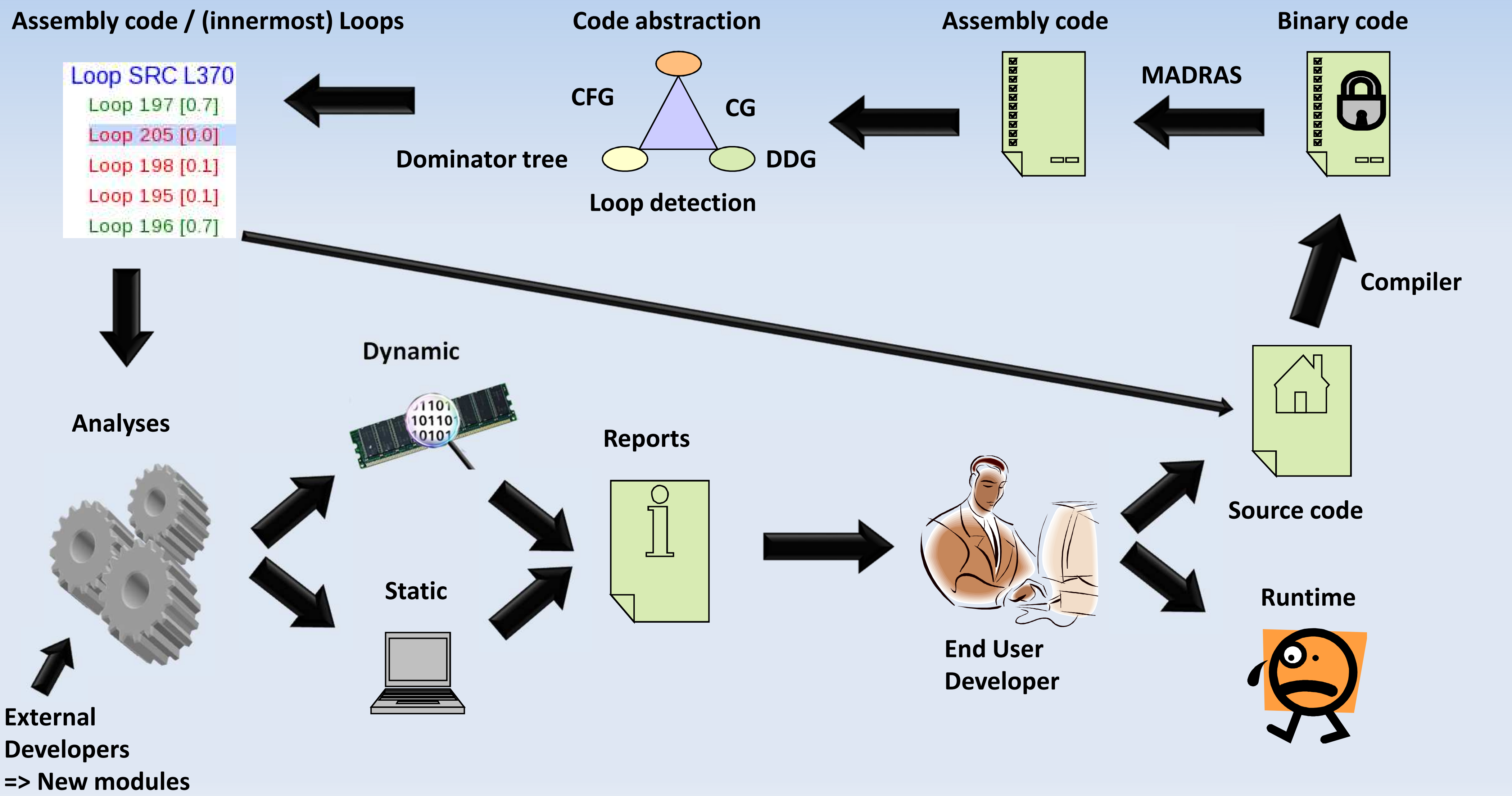
MAQAO Tool

- Built on top of the Framework
- Exploit existing framework features
- Produce reports
- Client/Server approach
  - User interface
  - Batch interface
- Loop-centric approach
- Packaging : ONE (static) standalone binary

MAQAO Tool overview

Modular Assembler Quality Analyzer and Optimizer

www.maqao.org



Static analysis

- Static performance model : STAN
  - Loop-centric
  - Predict performance
  - Take into account microarchitecture
  - Assess code quality
    - Degree of vectorization
    - Impact on micro architecture

Static analysis
The STAN module

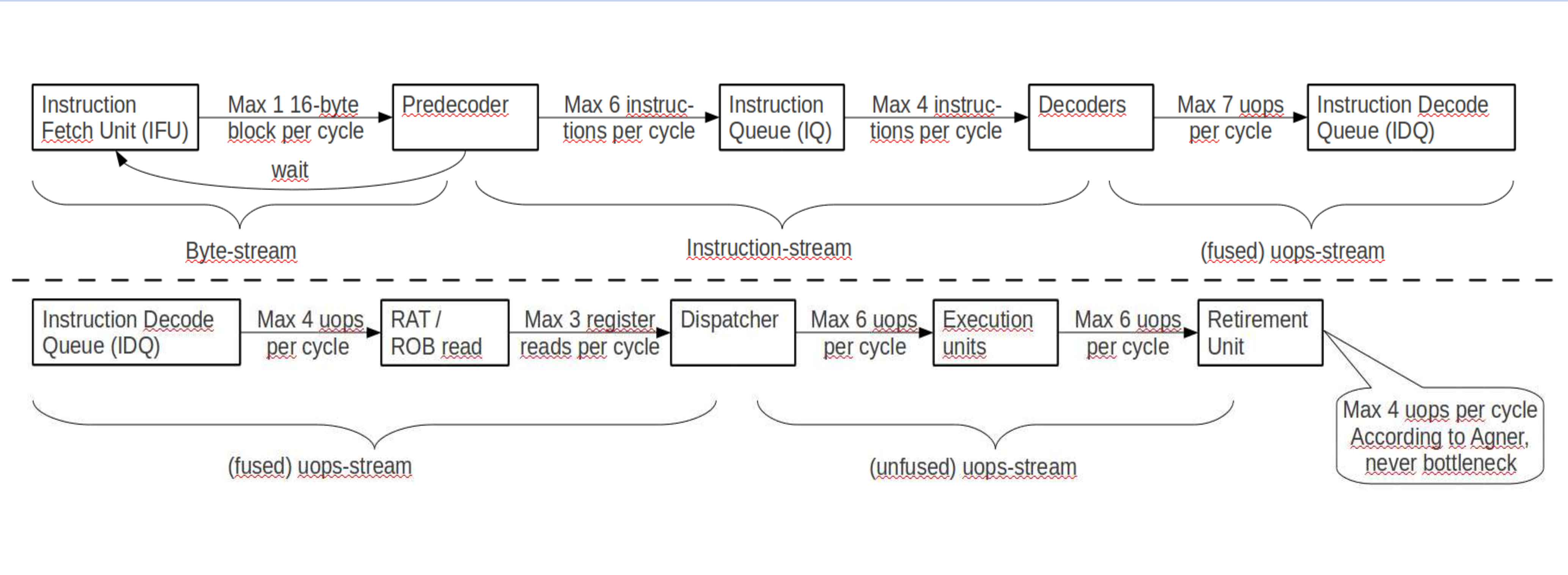
- Input
  - Micro-architecture (machine model)
  - Path to a binary file
  - Name of a function
- Output
  - CSV file
  - TXT file
- Analysis of all innermost loops in a given function
- STAN is also available via a MAQAO function

Static analysis
The STAN module

- 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 is fitted to the uarch)
  - Static performance (lower bounds on cycles)
  - Hints and workarounds to improve static performance

Static analysis

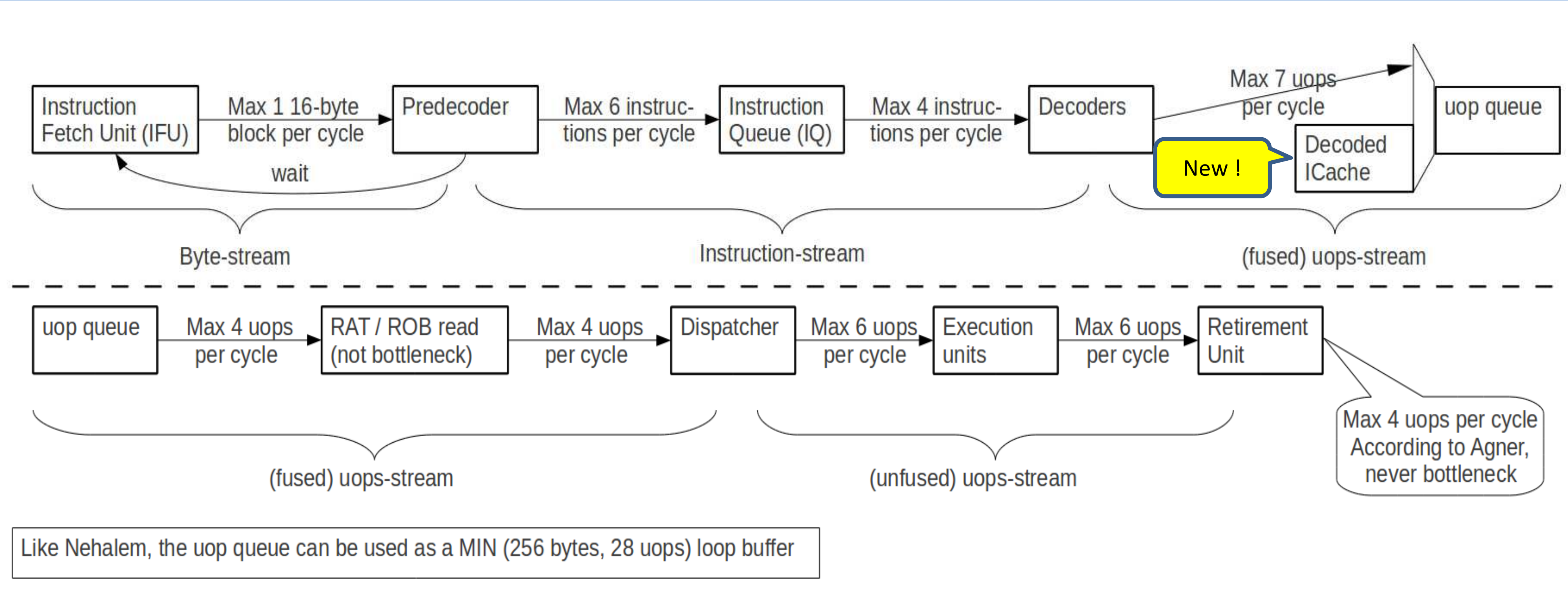
Core 2 and Nehalem Pipeline Model



On Core 2, IQ can be used as a MIN (64 bytes, 18 instructions) loop buffer
On Nehalem, IDQ can be used as a MIN (256 bytes, 28 uops) loop buffer

Static analysis

Sandy Bridge Pipeline Model



1.5 Kuops cache (100% hits for hotspots and 80% hits avg.)
Two 128 bits load units instead of one for Core 2 and Nehalem
AVX instructions set (vector size doubled compared to SSE)

Static analysis

Key analysis/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

TXT high level output example (1/2)

```
void div (int n, float a[n], float b[n]) {
  int i;

  for (i=0; i<n; i++)
    a[i] /= b[i];
}
```

```
MOVSS 0(%RSI,%RAX,4),%XMM0
DIVSS 0(%RDX,%RAX,4),%XMM0
MOVSS %XMM0,0(%RSI,%RAX,4)
ADD   $0x1,%RAX
CMP   %EAX,%EDI
JG    10
```

```
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

TXT high level 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).
```

Static analysis
TXT low level output example (1/2)

```
*************************************************
                Processing loop 0
*************************************************
Function: div
Source file: /tmp/test_newton_raphson.c
Source line: 67
Address in the binary: 10

*************************************************
                  Assembly code
*************************************************
MOVSS     0(%RSI,%RAX,4),%XMM0
DIVSS     0(%RDX,%RAX,4),%XMM0
MOVSS     %XMM0,0(%RSI,%RAX,4)
ADD       $0x1,%RAX
CMP       %EAX,%EDI
JG        10

*************************************************
             General loop properties
*************************************************
nb instructions    : 6
nb uops            : 6
loop length        : 23
used xmm registers : 1
used ymm registers : 0

Pattern: SS
nb instructions:
SS 3
```

```
nb FP arithmetical operations:
div 1

Bytes loaded: 8
Bytes stored: 4
Arith. intensity (FLOP / ld+st bytes): 0.08

Unroll factor: 1 or NA

FIT IN UOP CACHE

*************************************************
                    Dispatch
*************************************************
       P0    P1   P2   P3   P4   P5
Uops   1.33  1.33 1.50 1.50 1.00 1.33
Cycles 14.00 1.33 1.50 1.50 1.00 1.33

*************************************************
              Vectorization ratios
*************************************************
All       : 0%
Load      : 0%
Store     : 0%
Mul       = NA (no mul SSE or AVX instructions)
add_sub   = NA (no add_sub SSE or AVX
instructions)
Other     : 0%
```

Static analysis

TXT low level output example (2/2)

```
************************************************
                If all data in L1
************************************************
cycles: 14.00
FP operations per cycle: 0.07 (GFLOPS at 1 GHz)
instructions per cycle: 0.43
bytes loaded per cycle: 0.57 (GB/s at 1 GHz)
bytes stored per cycle: 0.29 (GB/s at 1 GHz)
bytes loaded or stored per cycle: 0.86 (GB/s at
1 GHz)
Cycles if fully vectorized: 3.50
Cycles executing div or sqrt instructions: 10-14
(second value used for L1 performances)
************************************************
                       End
************************************************
Loop ending at source line 7 is not unrolled or
unrolled with no peel/tail code
```

Vtune – MAQAO analysis coupling
(on going experimentation)

- MAQAO: static analysis with the STAN module
  - For instance, provides lower bound on cycles per iteration and vectorization ratio
- VTune: dynamic analysis, using sampling and thread profiling
- Correlating both analysis allows to:
  - Dynamic/static cycles = potential speedup factor
  - Refine understanding of memory bottlenecks
    - For instance, cacheline usage
  - Advise the user some optimizations (vectorization...)

Dynamic analysis

- Static analysis is optimistic
  - Data in L1$
  - Believe architecture
- Get a real image
  - Coarse grain : find hotspots
  - DECAN : compute / memory bound
  - MIL : specialized instrumentation

MIL : Instrumentation Language

- Why ? Yet another language ?
  - Need to handle coarse and fine grain issues
  - Tool to express such queries
  - DSL : Sufficiently rich for instrumentation purposes
  - Fast prototyping
  - Focus on what (research) and not how (technical)
  - Explore code properties (side effect)
  - What about OpenMP/MPI ?

MIL : Instrumentation Language

- Global variables
- Events
- Filters
- Actions
- Configuration features
  - Output
  - Language behavior (properties)

MIL : Instrumentation Language

- Probes
  - External functions
    - Name
    - Library
    - Parameters : int,string,macros,function
    - Return value
    - Demangling
    - Context saving
  - ASM inline : handles loops
  - Runtime embedded code (lua code within MIL file)

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

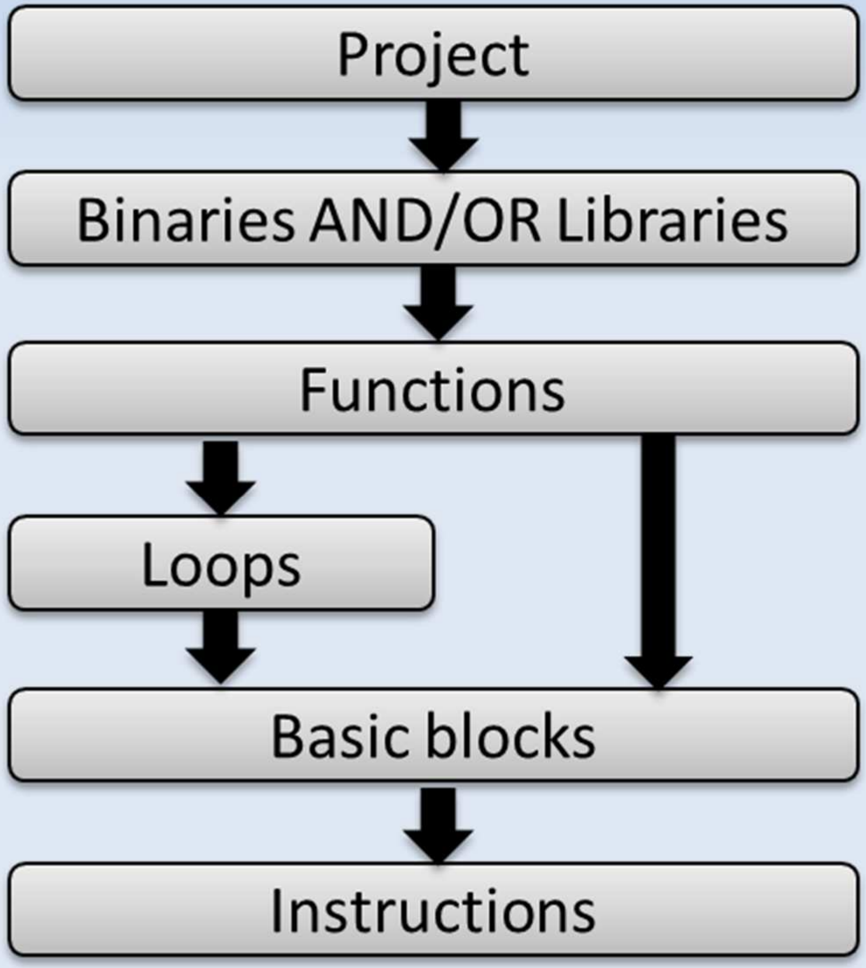
```
_ZN3MPI4CommC2Ev
MPI::Comm::Comm()
```

MIL : Instrumentation Language

- Events
  - Program : Entry/Exit (avoid LD + exit handlers)
  - Functions : Entries/Exits
  - Loops : Entries/Exits/Backedge
  - Blocks : Entries/Exits
  - Instructions : Before/After
  - Callsites : Before/After

MIL : Instrumentation Language

- Events : Hierarchical evaluation



MIL : Instrumentation Language

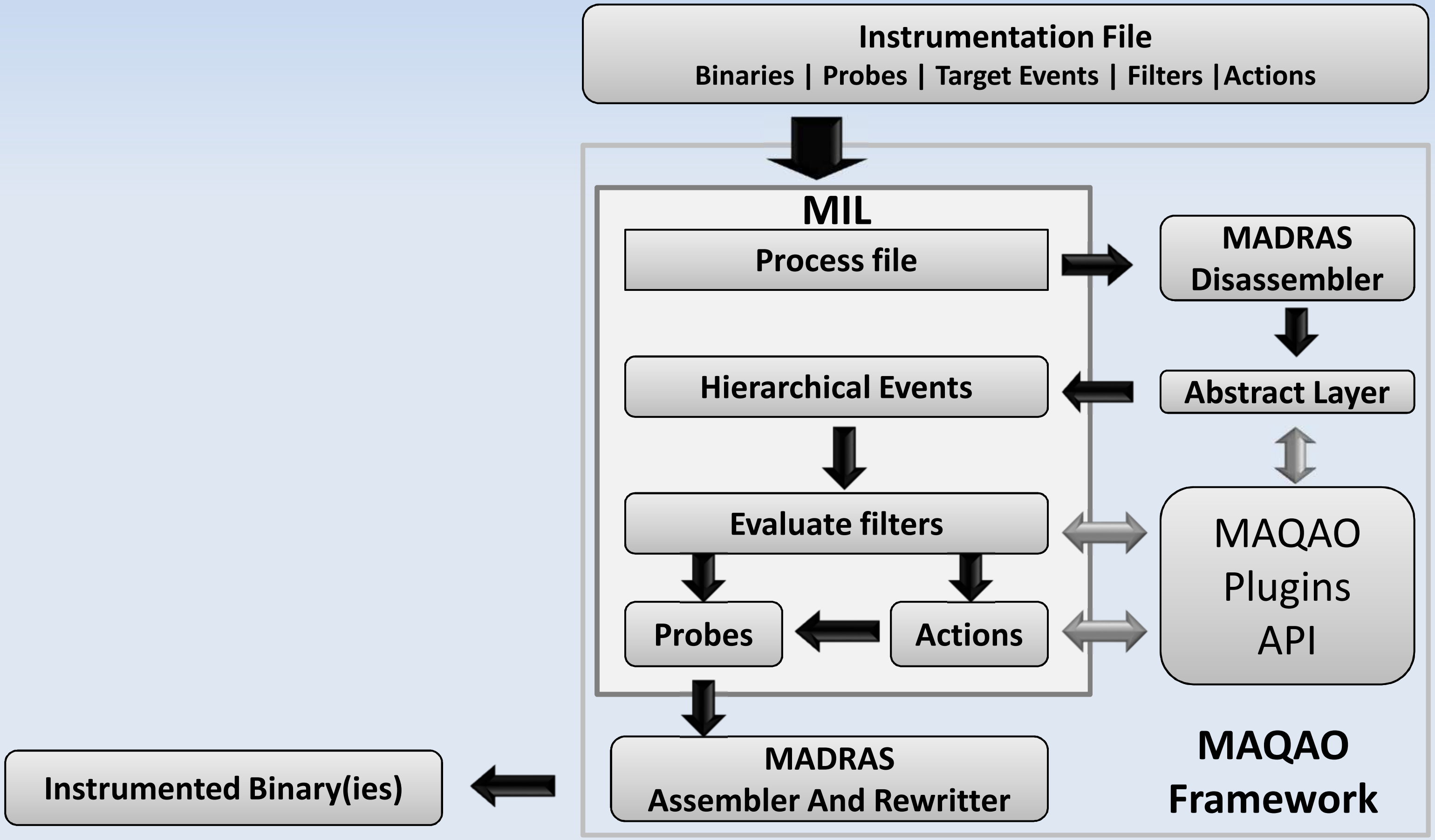
- Filters
  - Why ?
  - Lists : whitelist / blacklist (int,string,regexp)
  - Built-in : structural properties attributes (nesting level for a loop)
  - User defined : an actions that returns true/false

MIL : Instrumentation Language

- Actions
  - Why ? For complex instrumentation queries
  - Access to MAQAO Plugins API (existing modules)
  - Scripting ability (Lua code)
  - Function receives : event,patcher,gvars objects
  - User filters may be used to express very complex constraints (for instance based on static analysis)

MIL : Instrumentation Language

Another way to use the MAQAO Framework :
DSL for Building performance evaluation tools



Conclusion

- Select a consistent methodology
- Assess code quality through static analysis
- Detect hotspots
- Iterative approach to solve finer grain issues
- If no relevant existing module : use MIL

Thanks for your attention !

Questions ?

Setup

- Copy maqao_exercices.tar.bz2 from /tmp
- Extract + cd exercises
- Run : ”source env.sh”
- 4 folders containing each one exercice
  - memory
  - gvars
  - standalone_profiler
  - stan

MIL module : Getting started

- To invoke MIL module :
  - maqao module=mil input=MIL_FILE
- Run this command in each exercise folder and
- Given an exercise foldern replace MIL_FILE by the file finishing with .mil suffix
- Ex :
  - ”cd memory”
  - ”maqao module=mil input=load_store_mrt.mil”
  - ./mem_i

MIL module : file layout

- Helper code section (action,filters,...)
- Runtime code section (milRT class)
- Data section (reserved mil.data namespace)
- Declare section (external functions at Runtime)
- Global variables
- Global blacklist
- Events table
- Post instrumentation callback : at_instru_exit

Exercices : Outline

- Ex1 : looking for load/store operation
- Ex2 : using global variables
- Ex3 : a standalone simplified function profiler
- Ex4 : a specific loop profiler for STAN module
- Ex5 : using STAN module

Ex1 : looking for load/store operation

- In memory folder run :
  - maqao module=mil input=load_store_mrt.mil
  - ./mem_i
- This example shows how to instrument specific instructions : load ans stores using MAQAO Lua API
- This is done with user defined filters and MAQAO Lua API to determine if the instruction is a load or a store
- Prints selected instructions at runtime

Ex2 : using global variables

- In gvars folder run :
  - maqao module=mil input=gvars.mil
  - ./gvars_i
- In this exercise we will see how to insert calls to external functions and use global variable
- This example shows how to initialize a datatructure in an external function, keep the returned pointer and use it in further external calls.
- Source code of the patched binary and the external library can be found in src folder

Ex3 : a standalone simplified function profiler

- In standalone_profiler folder run :
  - maqao module=mil input=simple_function_profiler.mil
  - export OMP_NUM_THREADS=2 && ./bt.S.milrt
- In this exercise we will build a simple function profiler (aggregate time)
- In this example NPB-OMP bt.S (ICC compiled) binary will be used.
- We will use embedded runtime code so that the whole profiler is written in MIL.
- Prints results for each thread

Ex4 : specific loop profiler for the STAN module

- In stan folder run:
  - `make`
  - `maqao module=mil input=mil_get_loop_cycles.lua`
  - `maqao module=mil input=mil_get_loop_iters.lua`
  - `./my_div_baseline_inst_cycles 100000 2000`
  - `./my_div_baseline_inst_iters 100000 2000`
  - `maqao print_estimated_cycles.lua uarch=NEHALEM bin=my_div_baseline`
- In this exercise we will:
  - Instrument the my_div_baseline binary to get cycles and iterations number for innermost loops of the my_div function
  - Run the instrumented binaries
  - Run a MAQAO script to display, for each loop, the average iteration number and the estimated (using STAN) and measured number of cycles per iteration

Ex5 : using the STAN module

- How can I use STAN to improve my code quality ?
- Compiling C = A / B (vector notation) with gcc -O2
- The instrumentation process previously presented provides, for each innermost binary loop in the hottest function (my_div):
  - cycles per iteration (useful to compare with STAN)
  - number of iterations (useful to identify peel/tail loops)
- To analyse my_div_baseline with STAN:
  - `maqao module=stan uarch=NEHALEM bin=my_div_baseline fct=my_div lvl=2`

Ex5 : using the STAN module
baseline, -O2

- 1 binary loop, source loop not unrolled/vectorized:
  - `**not unrolled** or unrolled with no peel/tail code (including vectorization)`
  - `Your loop is **not vectorized**`
  - `The binary loop is composed of 1 FP arithmetical operations:`
  - ` - **1: divide**`
- STAN advises to compile with -O3:
  - `Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED...`
  - `**gcc: use O3** or Ofast`

Ex5 : using the STAN module
vectorized, -O3

- 3 binary loops, source loop was vectorized:
  - `It is (...)` **`unrolled by 4 (including vectorization)`**
  - `Your loop is` **`fully vectorized`**
- STAN detected a pathological case but gives no solution/hint (you can read “Bottlenecks”):
  - `Detected EXPENSIVE INSTRUCTIONS...`
- STAN advises to compile with special options to issue faster instructions:
  - **`gcc:`** `(`**`ffast-math`** `or Ofast) and` **`mrecip`**

Ex5 : using the STAN module

recip, `-ffast-math -mrecip`

- 1 binary loop, source loop not unrolled/vectorized:
  - `It is (...)` **`not unrolled`** `or unrolled with no peel/tail code`
  - `Your loop is` **`probably not vectorized`**
  - `The binary loop is composed of 5 FP arithmetical operations:`
  - **`- 1: fast reciprocal`**
- STAN advises to compile with -O3:
  - `Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED...`
  - **`gcc: use O3`** `or Ofast`

Ex5 : using the STAN module

unrolled, `-O2 -funroll-loops`

- 1 binary loop, source loop was unrolled by 8:
  - `It is (...) not unrolled or` **`unrolled with no peel/tail code`**
  - `Your loop is` **`not vectorized`**
  - `The binary loop is composed of 8 FP arithmetical operations:`
  - **`- 8: divide`**
- STAN computed a lower bound of 112/8 = 14 cycles per source loop iteration (no better than baseline):
  - `Assuming all data fit into the L1 cache, each iteration of the binary loop takes` **`112.00 cycles`**

Ex5 : using the STAN module

all_opt, `-O3 -ffast-math -mrecip`

- 1 binary loop, source loop was vectorized:
  - `It is (...)` **`unrolled by 4 (including vectorization)`**
  - `Your loop is` **`fully vectorized`**
  - `The binary loop is composed of 8 FP arithmetical operations:`
  - **`- 4: fast reciprocal`**
- STAN detected a pathological case but gives no solution/hint (you can read “Bottlenecks”):
  - `Detected EXPENSIVE INSTRUCTIONS...`

Ex5 : using the STAN module
Optimization speedup

- Baseline: ~14.0 cycles
- Optimized: ~2.3 cycles
- **Speedup: ~6.1x**
- **Still possible to go faster using STAN:**
  - Loop unrolling on the all_opt version to relax P5 execution port
  - Aligning arrays on 16B boundaries and inform the compiler about that to replace (MOVLPS, MOVHPS) “expensive” instructions pairs with MOVAPS