MIL

A language to build program analysis tools through static binary instrumentation

Andrés S. CHARIF-RUBIAL
achar@exascale-computing.eu

7ème rencontres de la compilation
4-6 December 2013

ECR Lab

- Joint Lab

- Research axes
  - Performance evaluation: MAQAO Tool
  - Application characterization
  - Energy
- We work with ISV partners
- Feedback from our user community

Introduction

Context

- Binary instrumentation: after compiler
- 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
  - Efficient: lowest overhead
  - Robust: ensure the program semantics
  - Accurate: correctly identify program structure

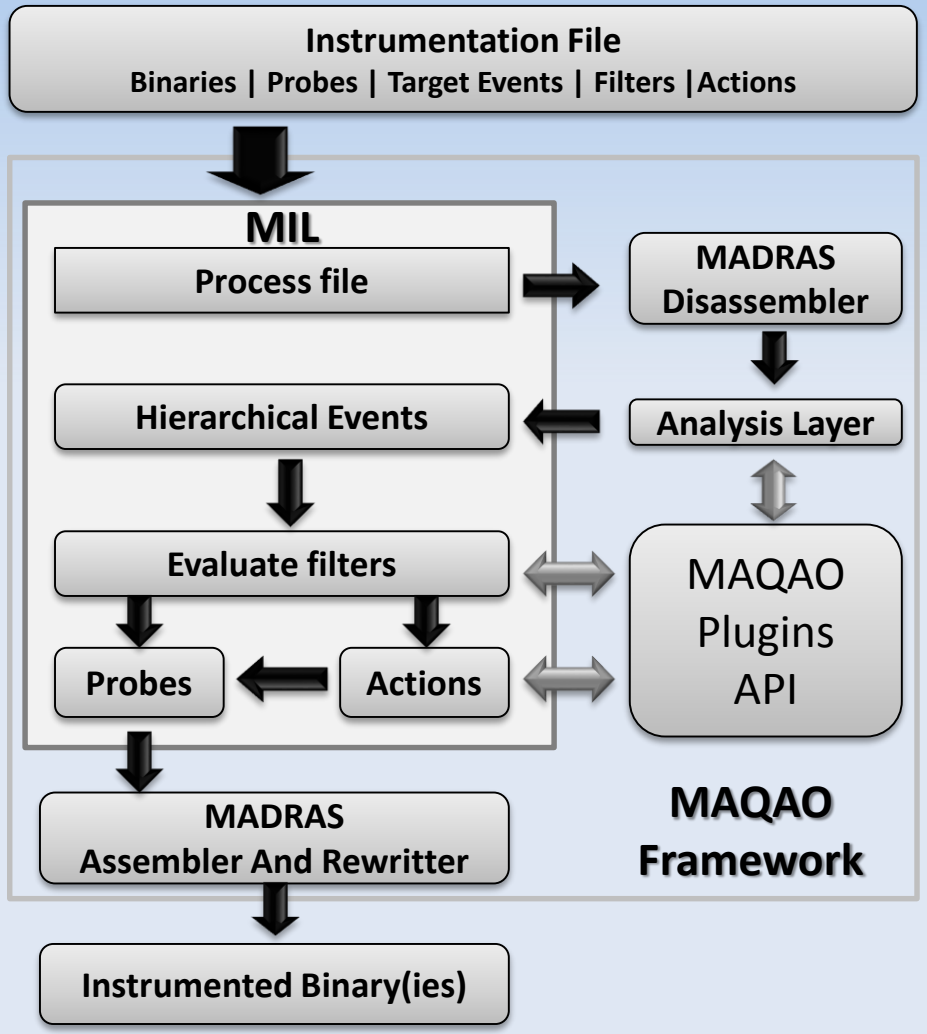
Introduction

Related work

	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

MAQAO Tool
Integration of MIL



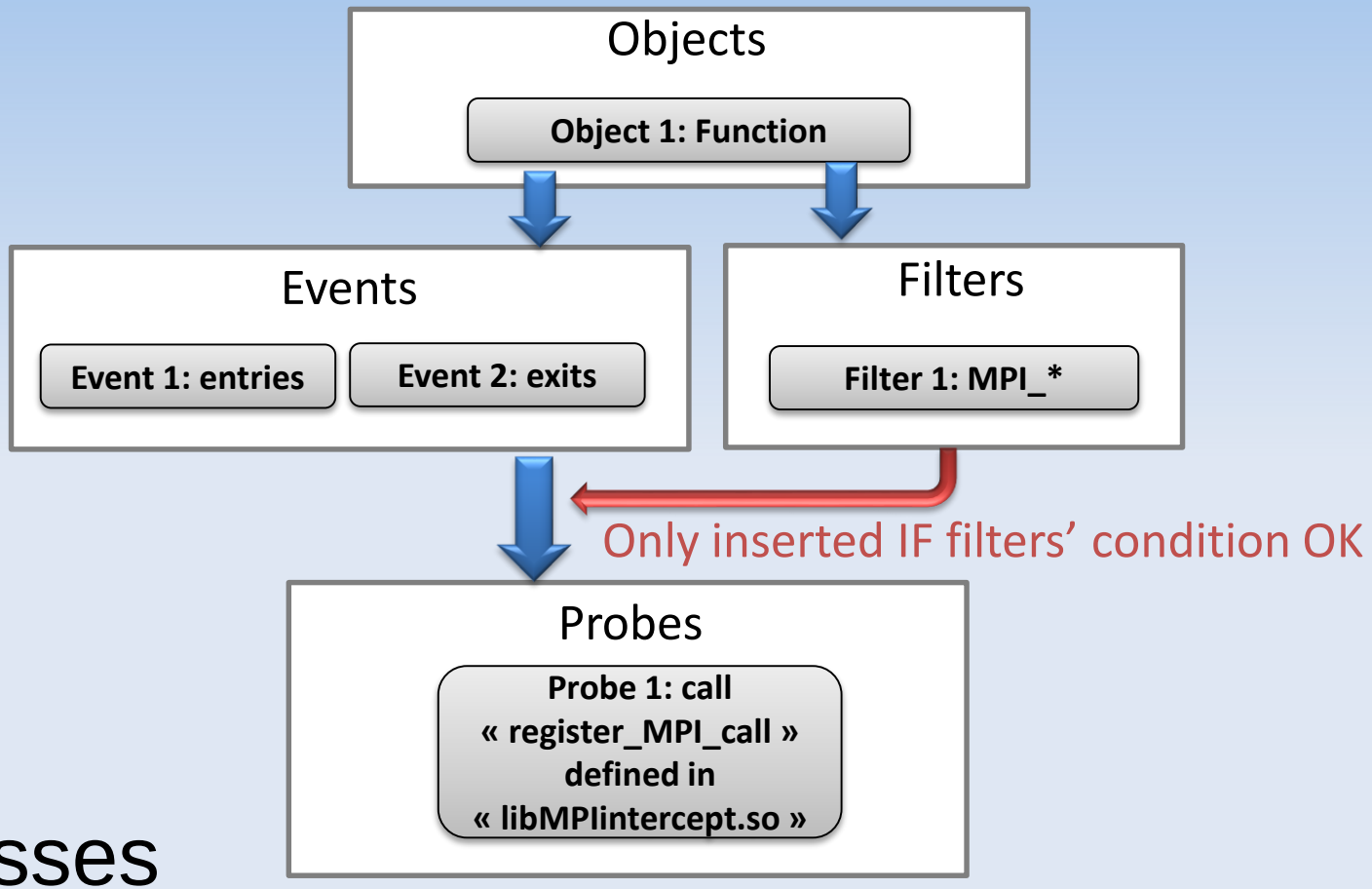
www.maqao.org

MIL: A language to build program analysis tools through static binary instrumentation.
In *20th Annual International Conference on High Performance Computing*, HiPC'13

Language concepts/features

Overview

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



Language concepts/features

Examples

Example 1: TAU Profiler

- Object
- Events
- Probes
- Configuration
- Comments

```
fct_iter = Iterator:new(0);

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);
```

Experiments

Comparing MIL and Dyninst overhead using TAU

Accuracy of results: output of thread1 for bt.A

MIL

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	1,164	1:07.796	1	1012	67796835	.TAU application
31.9	21,416	21,649	201	174096	107709	x_solve_omp
31.8	21,400	21,569	201	122492	107309	y_solve_omp
31.7	21,359	21,496	201	103416	106948	z_solve_omp
2.4	1,649	1,649	202	0	8167	compute_rhs_
0.2	156	156	201	0	776	add_

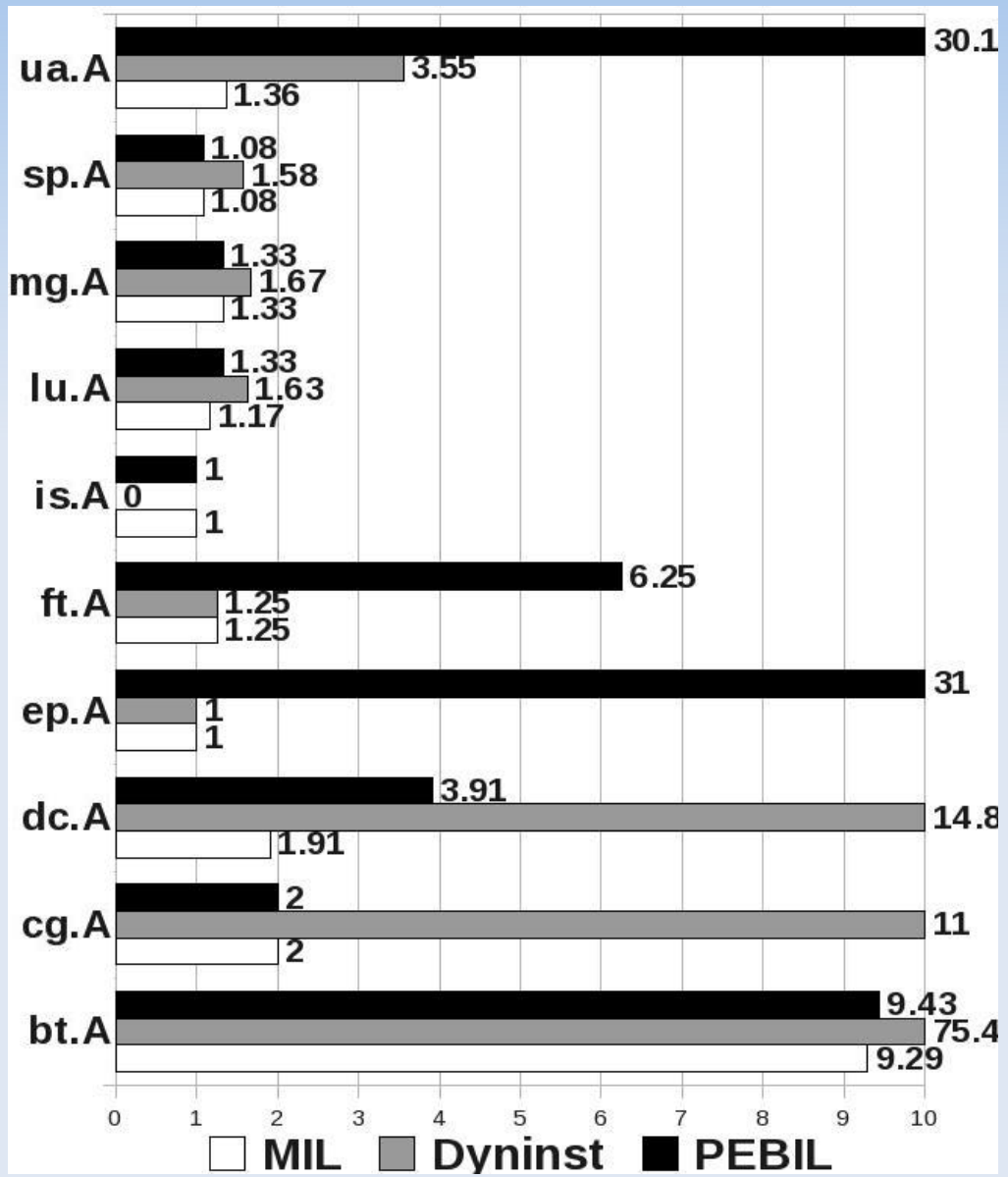
Dyninst

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	7,845	9:30.284	1	1012	570284826	.TAU application
32.6	3:04.814	3:05.867	201	155905	924715	void targ4161f9()
32.4	3:03.927	3:04.840	201	136524	919605	void targ419ff9()
32.4	3:03.162	3:04.547	201	207576	918145	void targ4146f8()
0.7	3,357	4,058	2	100001	2029312	void targ402c52()
0.3	1,763	1,763	202	0	8728	void targ40be37()

Experiments

Comparing MIL and Dyninst overhead using TAU

- Using TAU profiler
- NPB-OMP: 12 threads
- More robust: all
- Faster: up to 8x



1-Byte basic block problem — 8x

trampoline mechanism overhead — 4.5x, 8x

Collaborations

- Integrated into TAU toolkit (previous example)
  - tau_rewrite
  - More expressive:
    - MIL: 20 lines
    - Dyninst: 200 lines
- Ongoing integration with Score-P (H4H project)

Thanks for your attention !

Questions ?