Ph.D. Defense

On code performance analysis and optimization for multicore architectures

Andrés Salim CHARIF-RUBIAL

Advisors : William JALBY
Denis BARTHOU

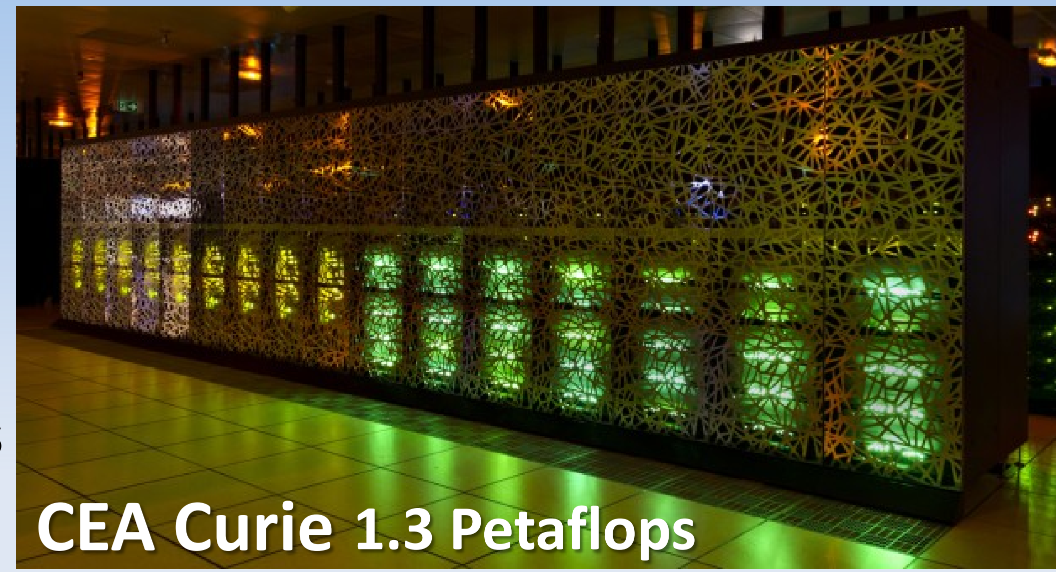
22 October 2012

Outline

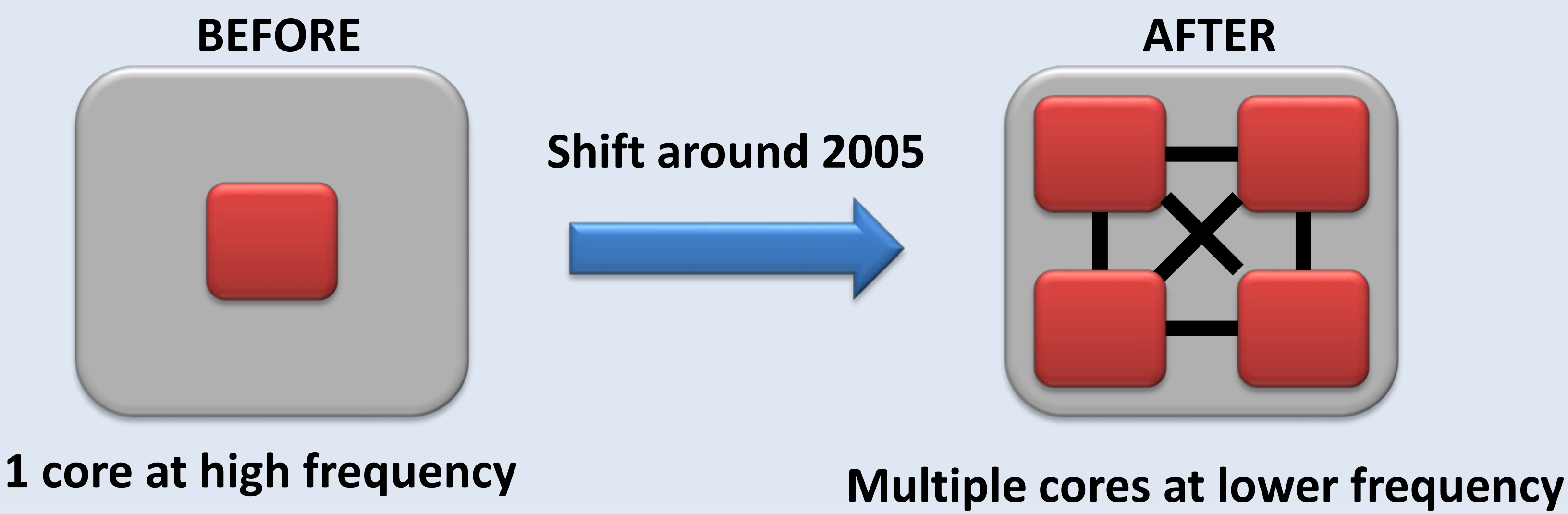
- Introduction
- Performance analysis
- Instrumentation Language
- Memory behavior characterization
- MAQAO tool
- Conclusion and Future work

Introduction
Context

- Deal with HPC applications
- Running on a cluster
  - Bigest: 16 Petaflops
  - 16 000 000 000 000 000 flops



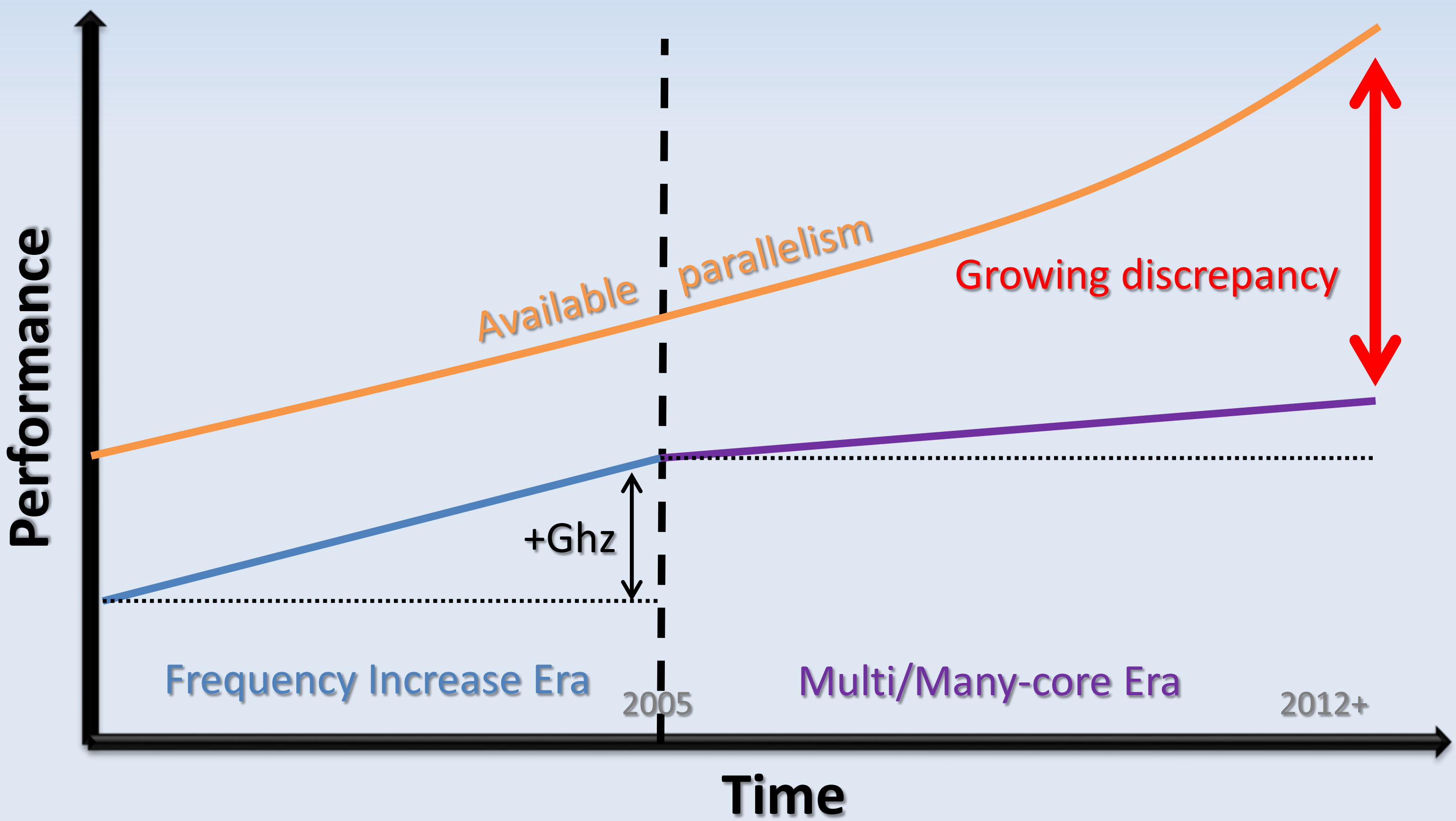
- Composed of multicore machines



Introduction

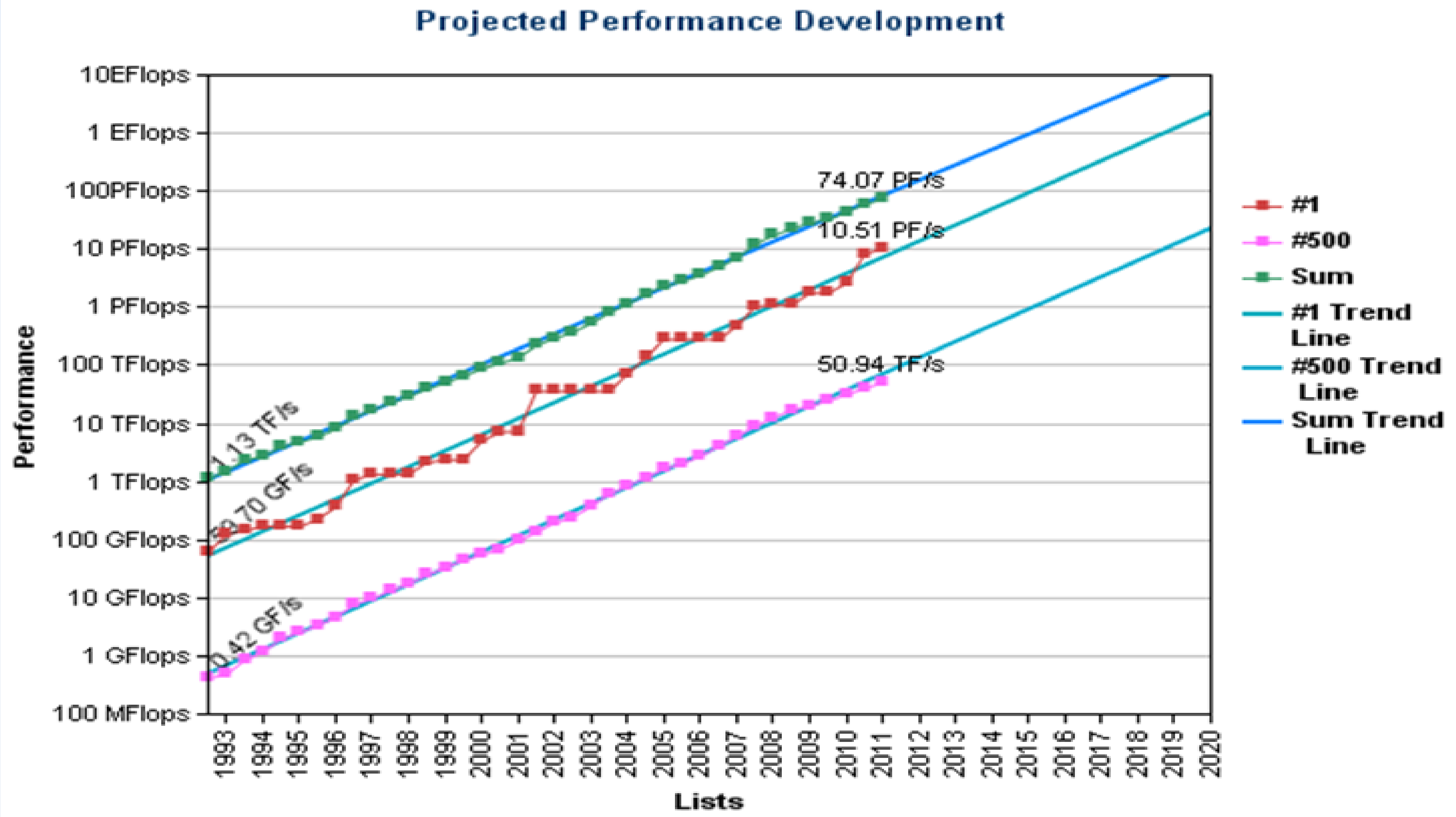
Leveraging parallelism

- Huge issue: exploiting parallelism



Introduction
Future trends

- Performance will continue increasing

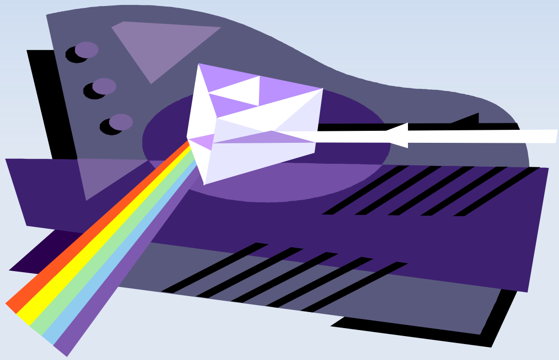


Outline

- Introduction
- Performance analysis
- Instrumentation Language
- Memory behavior characterization
- MAQAO tool
- Conclusion and Future work

Performance Analysis
What is it ?

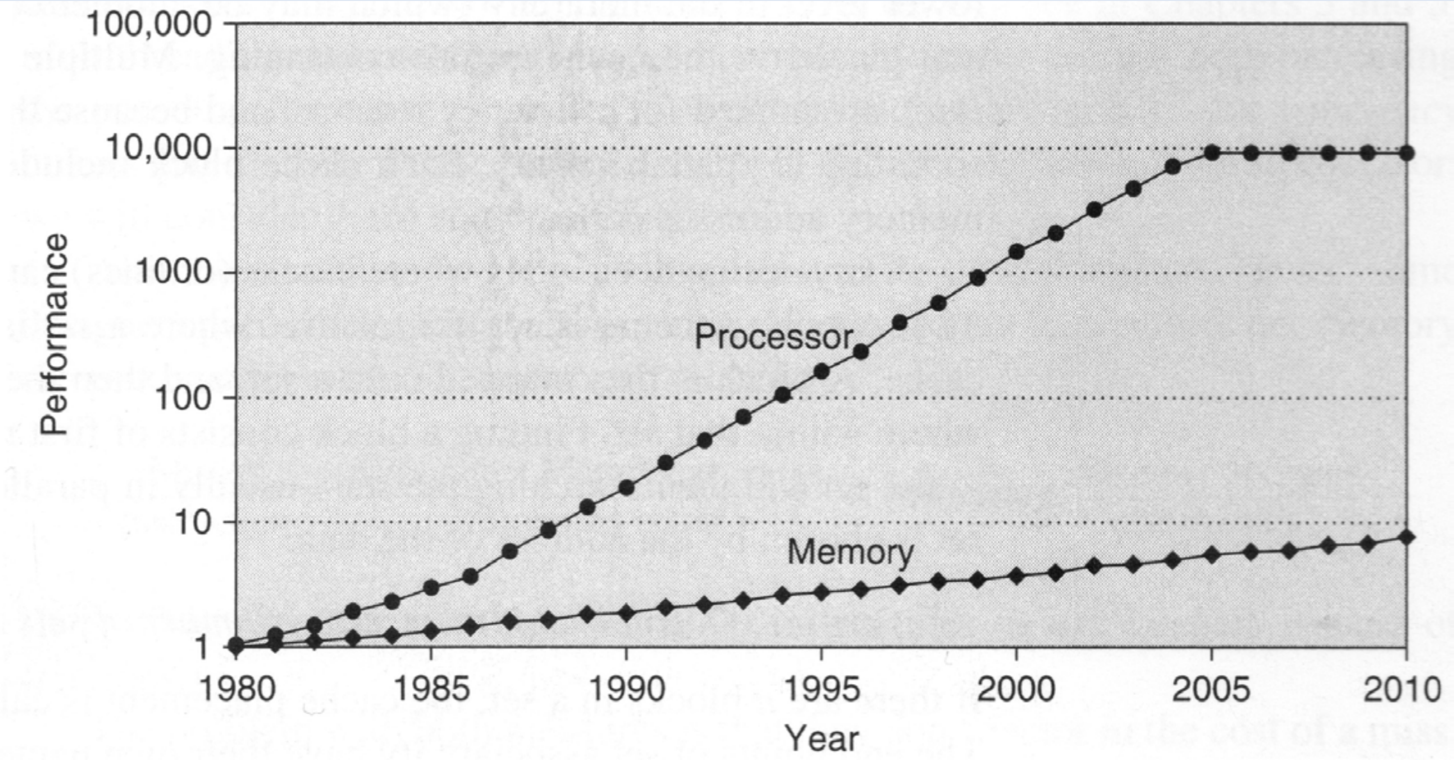
- Understand the performance of an application
  - How well it behaves on a given machine
- What are the issues ?



- Generally a multifaceted problem
  - Maximizing the number of views = better understand
- Use techniques and tools to understand
- Once understood → Optimize application

Performance Analysis
Why is it complex ? (1/4)
The memory wall

- Modern machines are very complex:
  - Complex architectures: not easy to fully exploit
  - Access to memory = huge impact: the memory wall



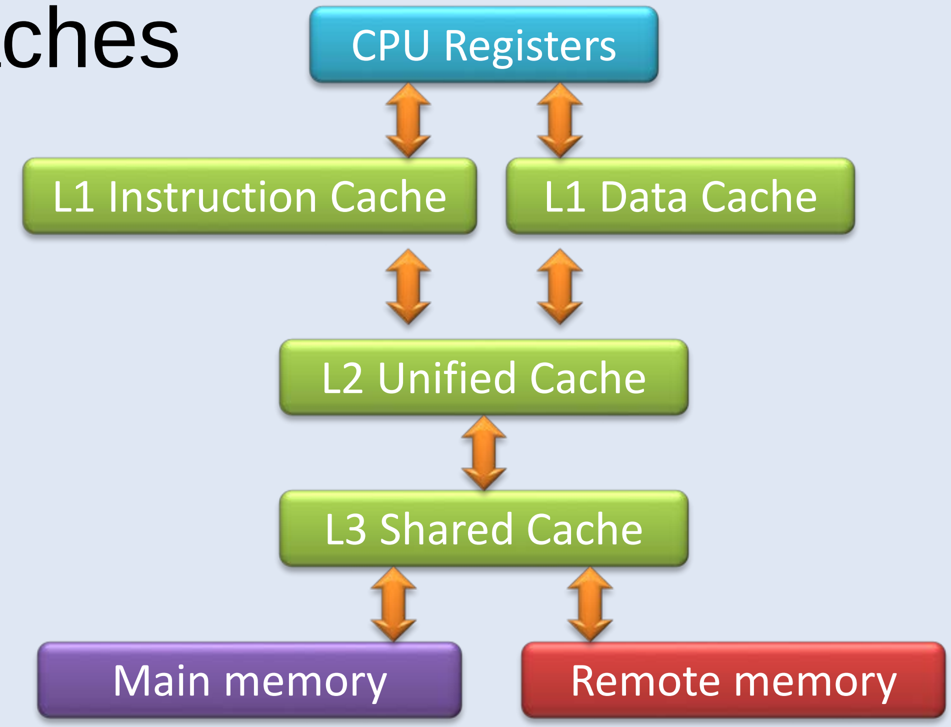
Performance Analysis

Why is it complex ? (2/4)

The memory wall

- A variable cost to access data

- To avoid memory cost: caches
- Issues related to caches:
  - Structure
  - Addressing: hit / miss
  - Data coherency
  - Data locality

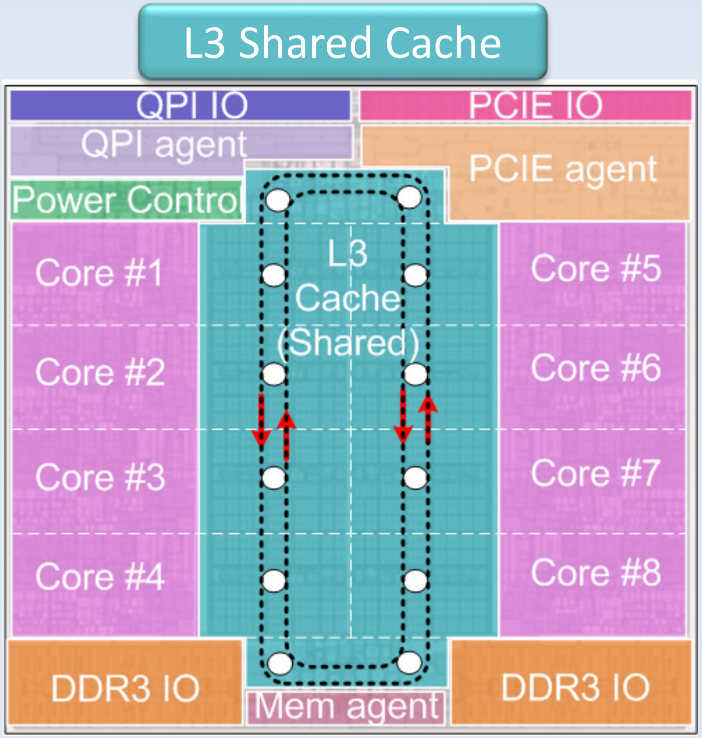


Performance Analysis

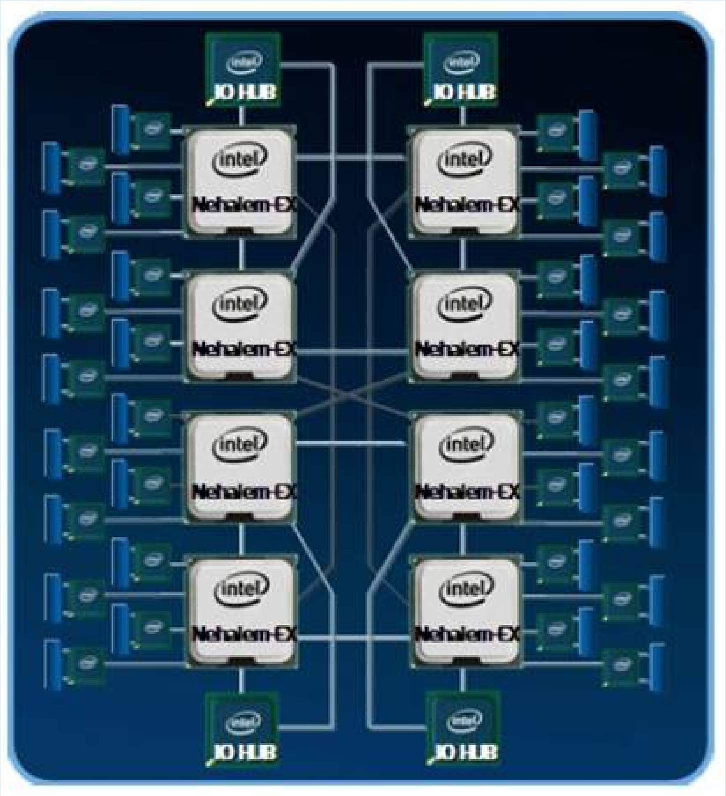
Why is it complex ? (3/4)

The memory wall

- More complex mechanism for LLC (NUCA)
- Remote memory location: NUMA



NUCA



NUMA

Performance Analysis
Why is it complex ? (4/4)

- Performance issues can occur at multiple levels:
  - Source | Compiler (Binary) | RT | OS | Hardware
- Too much is expected from the compiler
  - ”Usual” compilers: lack of a dynamic model
- Multiple parallel programming paradigms exist
  - Tools must take it into account
  - Generally we need multiple tools

Performance Analysis

Multiple analysis approaches

- Modeling:
  - \+ Fast
  - \- Low precision
- Simulation:
  - \+ Precise
  - \- Very slow
- Measurement :
  - Tracing: precise behavior ➡ Precise but slow
  - Sampling: rate or count ➡ Fast but less precise
  - Profiling: agregated statistics ➡ tracing, sampling

Performance Analysis
Existing tools

Classification / Tools	Analysis Method					Insertion Level		User feedback	
	Modeling	Simulation	Measure			Binary	Source code	Hints	Link to source
			Tracing	Profiling	Sampling				
gprof					X		X		
Cachegring Callgrind		X							
CMP$IM		X	X			X			
VTune					X	X			X
TAU			X	X		X	X		X
HPCToolkit					X				
Open|SpeedShop					X				
Scalasca			X	X		X	X		
SvPablo					X				X
PerfExpert								X	X
MAQAO		X	X	X		X	X	X	X

Performance Analysis

Target? Contributions

- Current tools not sufficient to fix memory issues
  - Need a precise memory behavior characterization
- Focus on one machine node
  - Shared memory model: OpenMP
- Helpfull analyses for users:
  - Provide usefull and understandable feedback
  - That correlates issues to source code
- Binary level: instrument OpenMP programs

Outline

- Introduction
- Performance analysis
- Instrumentation Language
- Memory behavior characterization
- MAQAO tool
- Conclusion and Future work

Instrumentation Language
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
Robust	Yes	Yes	No

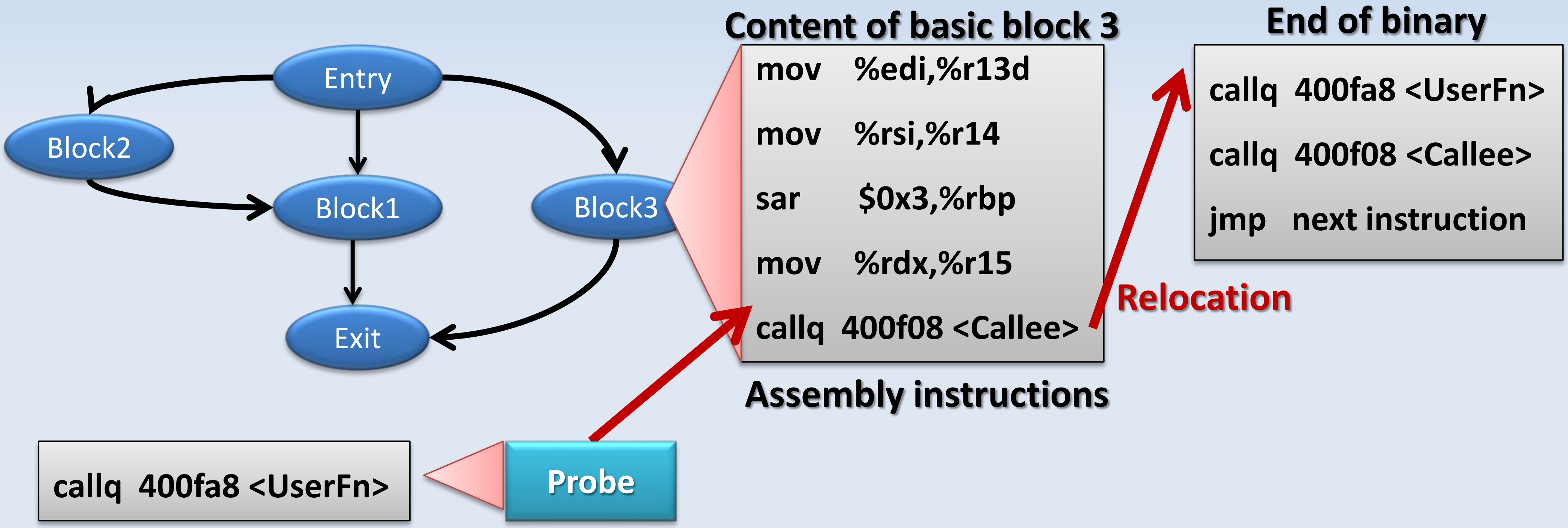
- Current state of the art:
  - Dyninst appears as the most complete
  - Not sufficient

Instrumentation language
Why? Yet another language ?

- 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
- Drive binary manipulation layer of MAQAO tool

Instrumentation Language
What is binary instrumentation ?

- Inserting probes at specific points
  - Example: before a call site



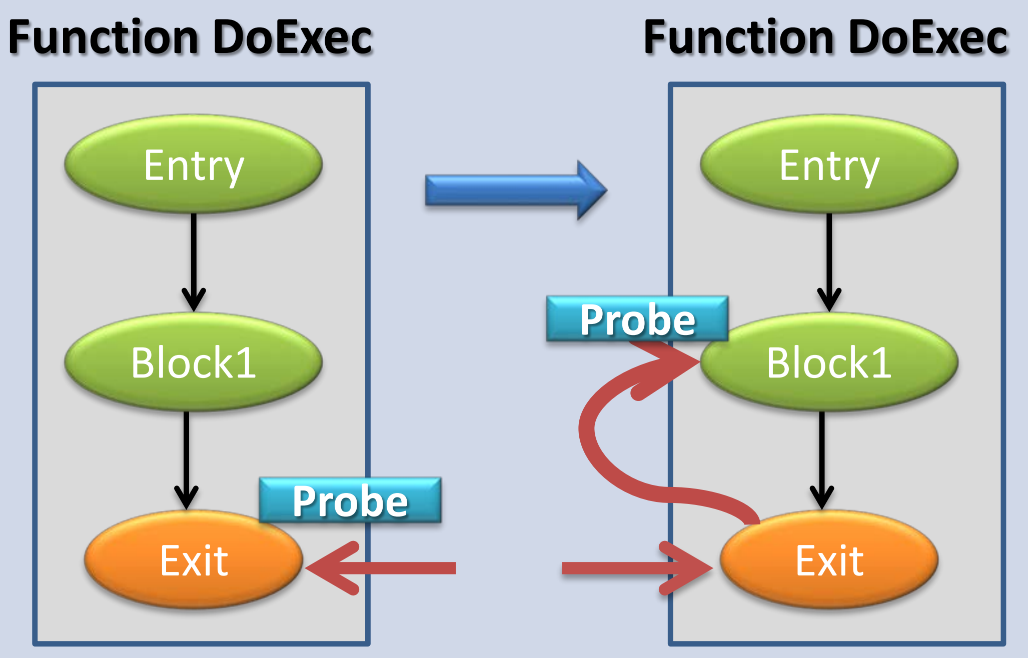
  - Using instruction or basic bloc relocation

Instrumentation Language

Advanced static analysis

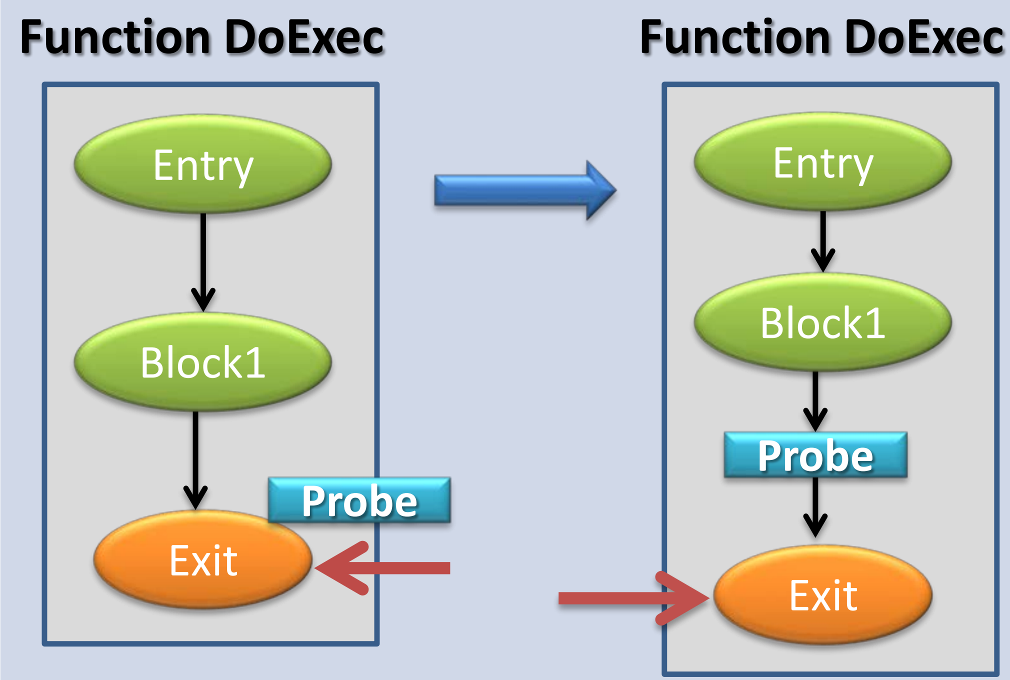
- Problem: instrumenting small basic blocks

Method 1: using trampolines



Cannot deal with 1-Byte basic blocks

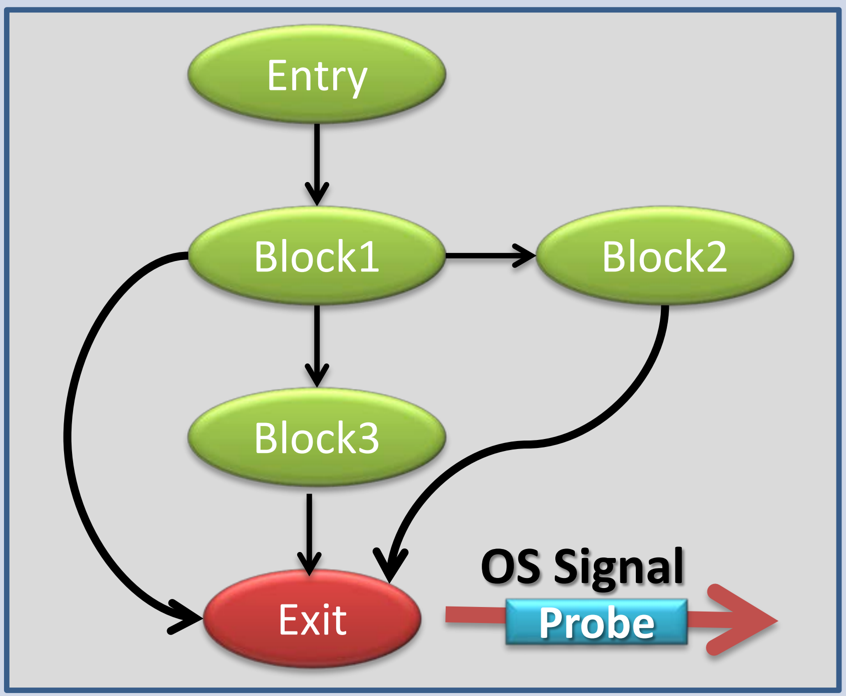
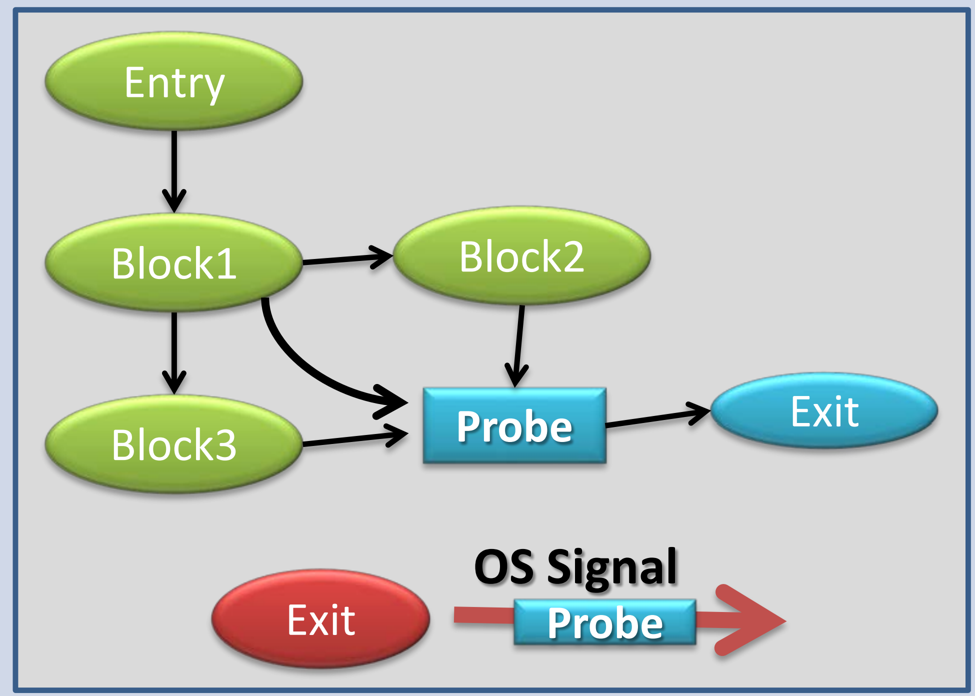
Method 2: function relocation



Cannot handle pointer (indirect branches)

Instrumentation Language
Advanced static analysis

- Problem: instrumenting 1-Byte blocks

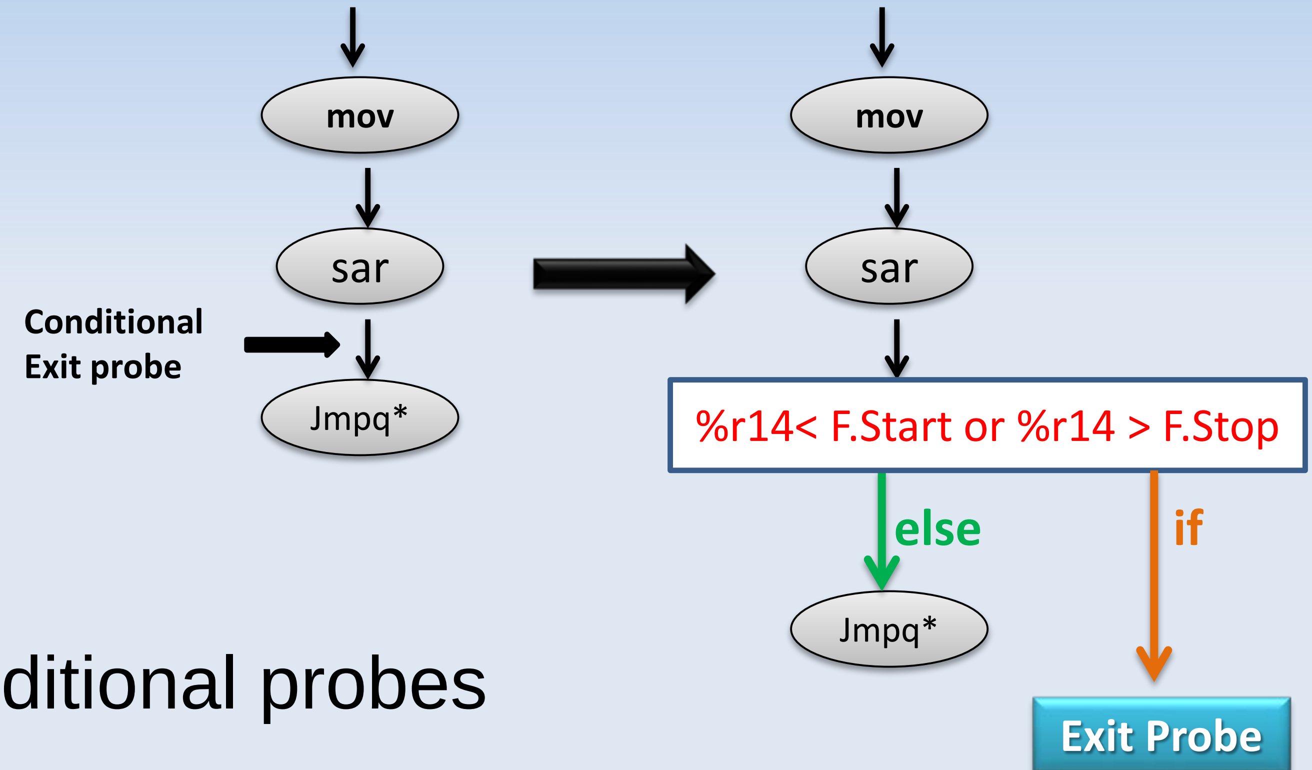
Method 1: only OS signal handlers	Our method: use predecessors (CFG)
Function DoExec 	**Function DoExec** 
Huge overhead	**Minimizes/Removes OS Signal execution**

- Exemple: dc.A (NPB-OMP) => 8x improvement

Instrumentation Language
Advanced static analysis

- Resolve indirect jumps: locate hidden exits

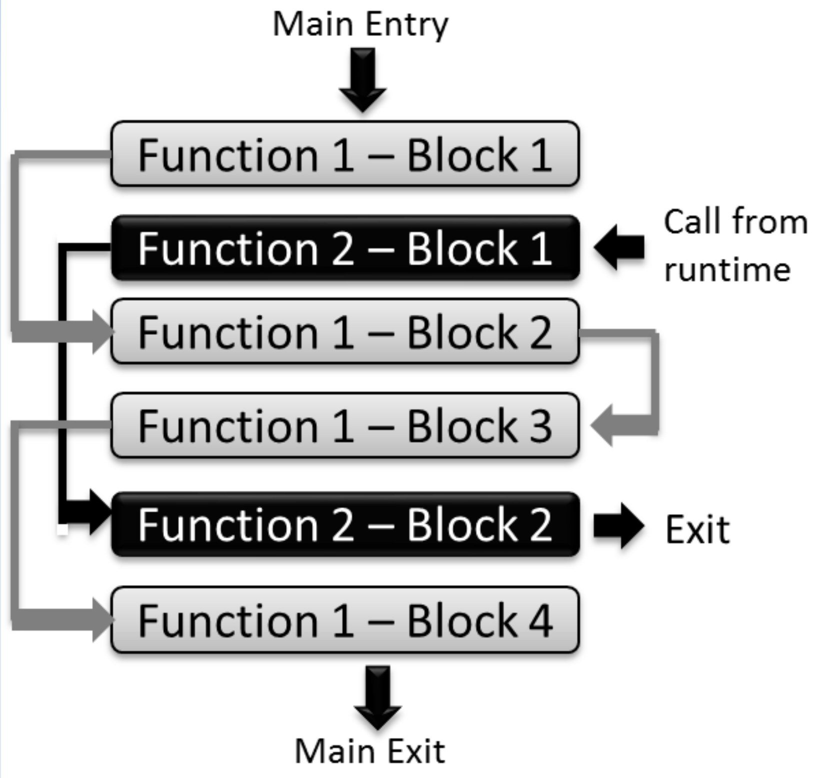
```
             ...
mov    %rdx,%r15
mov    %edi,%r13d
mov    %rsi,%r14
sar     $0x3,%rbp
jmpq  * %r14
             ...
```



  - Introduced conditional probes
  - Using ranges (function start/stop)
  - If so: insert exit probe(s)

Instrumentation Language
Advanced static analysis

- Handling interleaved functions
  - Required for OpenMP codes
  - Example: bt.A (NPB-OMP)
  - Solution:
    - Detect connected components (static analysis)
  - Try to detect inlining:
    - Heuristic: callsite + debug info
    - Works most of the time



Instrumentation Language

Overview

Instrumentation File

Binaries | Passes | Properties| Global variables | Probes | Events | Filters |Actions |Runtime code

↓

Instrumentation Language interpreter

Based on Lua language

↓

Modified binary(ies)/library(ies)

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
    - Parameters: int,string,macros,function (static⇔dynamic)
    - Return value
    - Demangling
    - Context saving
  - ASM inline: gcc-like
  - Runtime embedded code (lua code within MIL file)

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

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

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

Instrumentation Language

What does it look like ?

Ex: TAU Profiler

- Events
- Probes
- Configuration

```
run_dir = "/PATH_TO_OUTPUT_FOLDER/",
at_exit = {{ name = "tau_dyninst_cleanup " , lib = " libTau.so " }},
main_bin = {
  path= "/PATH_TO_main_binary",
  output_suffix = "_i",
  envvars="LD_LIBRARY_PATH=/PATH_TO_tau_library/",
  functions={{
    entries = {{
        at_program_entry = {{
            name = "trace_register_func",  lib = "libTau.so",
            params = {
               {type = "macro",value = "fct_info_summary"},
               {type = "macro",value = "profiler_id"},
            }
        }},
        name = "traceEntry", lib = "libTau.so",
        params = { {type = "macro",value = "profiler_id"} }
     }},
    exits = {{
        name = "traceExit",  lib = "libTau.so",
        params = { {type = "macro",value = "profiler_id"} }
    }}
  }}
};
```

Instrumentation Language
Collaborations

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

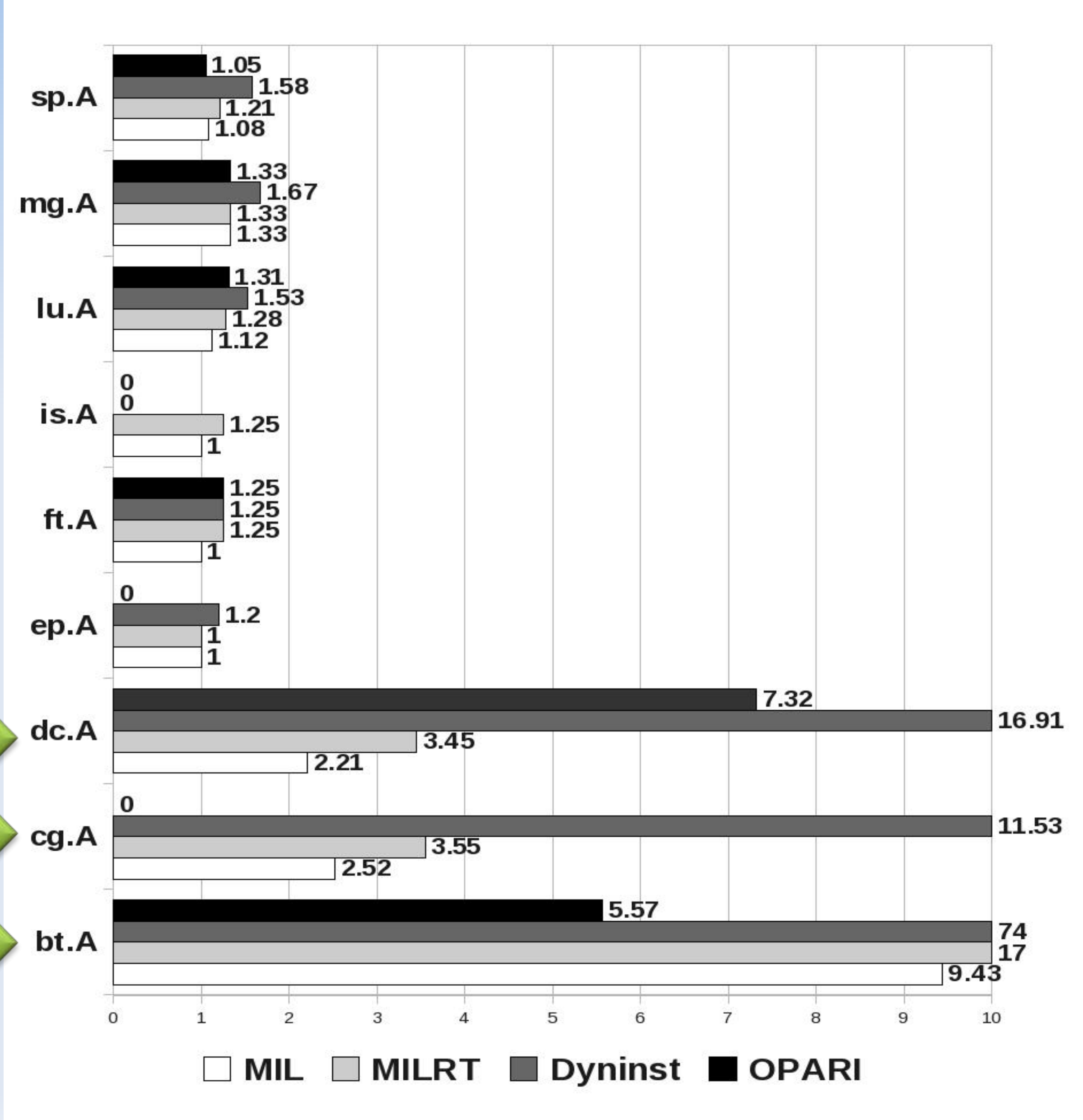
Instrumentation Language

Comparing MIL and Dyninst overhead using TAU

- Using TAU profiler
- NPB-OMP: 12 threads
- More robust: all
- Faster: up to 8x
- JIT version (MILRT) remains affordable

1-Byte basic block problem — 8x

trampoline mechanism overhead — 4.5x, 8x



Instrumentation Language

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()

Outline

- Introduction
- Performance analysis
- Instrumentation Language
- Memory behavior characterization
- MAQAO tool
- Conclusion and Future work

Memory behavior characterization

Overview

- Target: memory bounded applications
- Focus on OpenMP (2.5) applications
- A loop centric approach
- Tracing = 2 major challenges:
  - Storing all the memory addresses
  - Time to gather the trace
- Analyze the traces:
  - Single threaded: access patterns
  - Multi threaded: understanding interactions between threads

Memory behavior characterization

Storing all the memory addresses

- Targets memory instructions: loads, stores
- Per thread – Per instruction
- Trace collection: memory trace library (MTL)
  - Based on NLR algorithm (Ketterlin & Clauss)
  - Handles multi-threaded applications
  - Added simplified timestamps (cannot compress all timestamps)
- Simplified timestamps:
  - MIN-MAX intervals
  - Explicit synchronization: OpenMP = #OMP_BARRIER

Memory behavior characterization

Compressing address references

Source code

```
for (int n=0; n<M; n++)
 if (lambdax[n] > 0.)
  for (int i=0; i<NCz; i++)
   for (int j=1; j<NCx; j++)
    J_upx[IDX3C(n,j,M,i,(NCx+1)*M)] = …
```

Trace for store instruction

for i_0 = 0 to 49
 for i_1 = 0 to 63
  for i_2 = 0 to 149
   for i_3 = 0 to 198
    val 0x7f00bd1f0690 + 8*i_1 + 217600*i_2 + 1088*i_3

Start address
Offset
Level i_n

- Polytope model:
  - Compression: regular accesses are stored as loops
  - Do not represent source loop but spatial locality
  - Each level i_n: a different offset based on the same start address
- Strides can be easily derived:
  - For each level: stride = offset / sizeof(instruction)
- Each instruction can have multiple polytopes (regularity)

Memory behavior characterization

Instrumentation time

- Naive method:
  - instrument all memory accesses
- Enhanced method: prior static analysis
  - Find loop invariants and inductions
  - Instrument invariants
  - Ignore memory accesses based on them (derived)
  - Instrument naively all the others
  - Reconstruct address flows

Memory behavior characterization

Comparing Naive and Enhanced methods

- Dramatically improves performance in some cases
- Lowers, but cannot do much with irregular codes

Comparing instrumentation overheads

Benchmark	Naive Overhead	Enhanced Overhead	Improvement
SOMP 312.swim_m	273x	0.04x	6825x
SOMP 314.mgrid_m	974x	8.36x	116.5x
NAS PB ft.B	2160x	349x	6x

Memory behavior characterization

Exploiting the memory traces

- Single threaded aspects
  - Transformation opportunities, e.g.: loop interchange
  - Data reshaping opportunities , e.g.: array splitting
  - Detect alignment issues
- Understanding interactions between threads :
  - Load balancing
  - Reuse / False sharing
  - Thread affinity

Memory behavior characterization

Single threaded aspects: Inefficient patterns

Real code example: PNBENCH

- Application from CEA
- Parallel programming model: MPI
- Profiling with MAQAO tool provides hotspots:

Function	Loop (MAQAO id)	% of Wall time
flux_numerique_z	193	18
	195	
flux_numerique_x	204	17
	206	

- These loops where characterized as memory bounded
- Need a precise memory behavior characterization

Memory behavior characterization

Single threaded aspects: Inefficient patterns

Real code example: PNBENCH

```
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: PNBENCH

- 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**

Memory behavior characterization

Single threaded aspects: data alignment

- Instructions in original code not aligned:
  - Padding if complex structure
  - Compiler flags, pragmas to align (e.g.: vectors)
  - Allocate aligned memory: use posix_memalign()
- Architecture issue: even if aligned
  - Up to 10 cycles penalty
  - Micro benchmarking on each new machine
  - Warn user about values (alignment) to avoid

Memory behavior characterization

Understanding interactions between threads

Motivating example

- Using all the available theads is not always the best choice
  - Find out the best thread number

Benchmark	Reference		Best		Gain
	WTime (s)	Threads	WTime (s)	Threads	
NPB CG.A	0.62	96	0.42	36	32%
NPB FT.A	2.29	96	1.47	48	35%
SOMP 320.mgrid_m R	111.14	40	84.71	32	24%
SOMP 312.swim_m R	122.63	40	79.22	32	35%

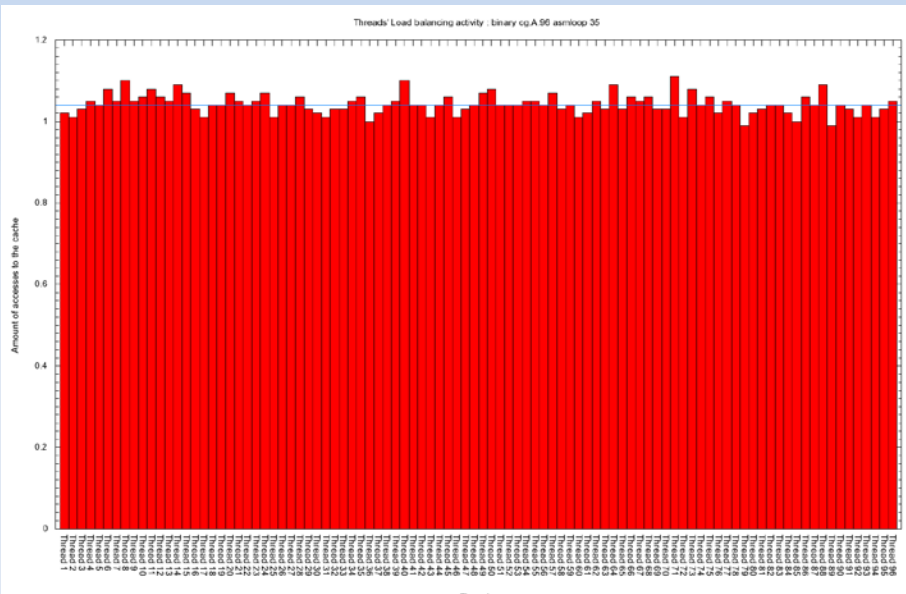
Memory behavior characterization

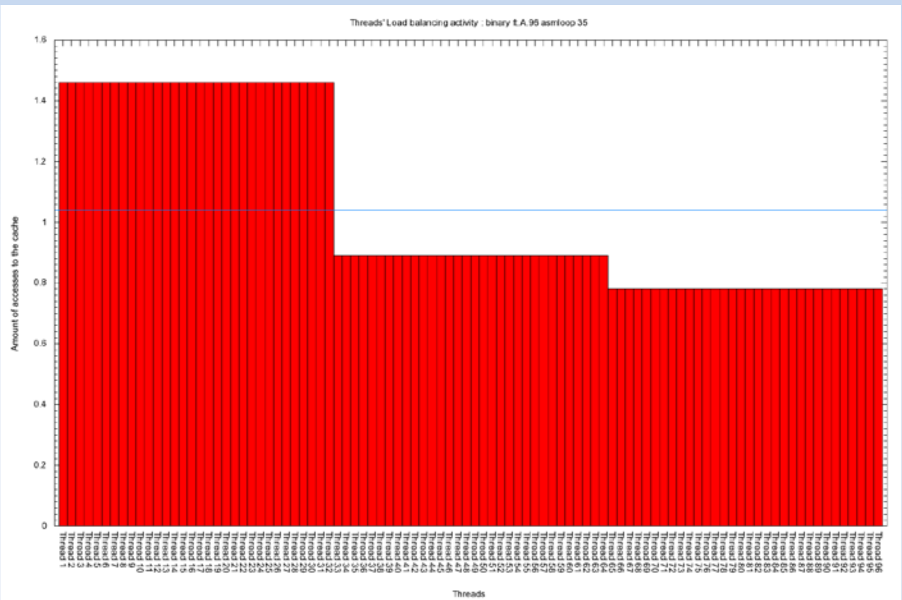
Understanding interactions between threads

Load balancing

CG (left) and FT (right) NAS Parallel benchmark running on 96 Threads

% memory accesses





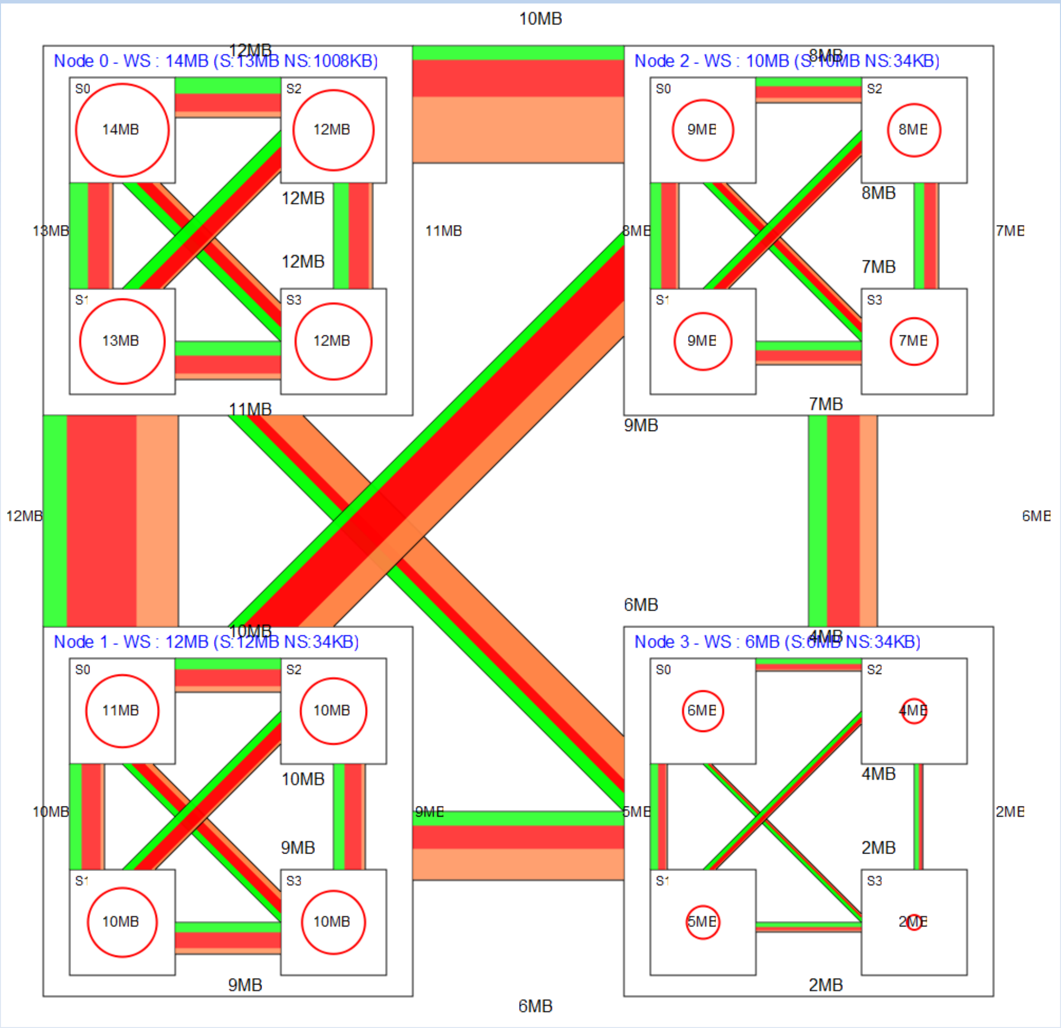
Threads

- Best execution time: 36 | 48 threads with a compact affinity.
- Not sufficient to understand

Memory behavior characterization

Understanding interactions between threads

Data sharing



Load/Load
Load/Store
Store/Store
Working set

LU decomposition application (OpenMP) on a 96 cores machine (4 nodes – 16 sockets)

Evaluates data sharing between Nodes/Sockets :

- Working set (shared/not shared)
- Coherence based on shared cache lines (worst case)

Memory behavior characterization

Understanding interactions between threads

Data sharing

- Rearranging threads: different pinning (affinity)
  - Automatically : find and swap candidates
  - Let user choose
- Reduce the number of thread
  - Shared resources saturation
  - Lack of parallelism (communication waste)
- Predict behavior on next generation architectures
  - Add architecture definitions
  - Generate corresponding trace on existing architectures

Memory behavior characterization
Understanding interactions between threads
Results

- OpenMP runtime parameters
  - Available metrics not sufficient to predict the correct number of threads
  - Suspect resource saturation issue when using all the available threads
  - Affinity proposed by Intel runtime provides close to best results
  - Symmetrical nature of OpenMP codes is an issue

- Reuse / False Sharing
  - Benchmarks does not exhibit significant issues due to false sharing
  - Maybe more in real applications

Outline

- Introduction
- Performance analysis
- Instrumentation Language
- Memory behavior characterization
- MAQAO tool
- Conclusion and Future work

MAQAO Tool

Overview

- Binary level : what is really executed
- Loop-centric approach
- Correlate binary to source code
- Coupling static and dynamic analyses
- Produce user-understandable reports
- Iterative approach
- Extensible through a scripting interface

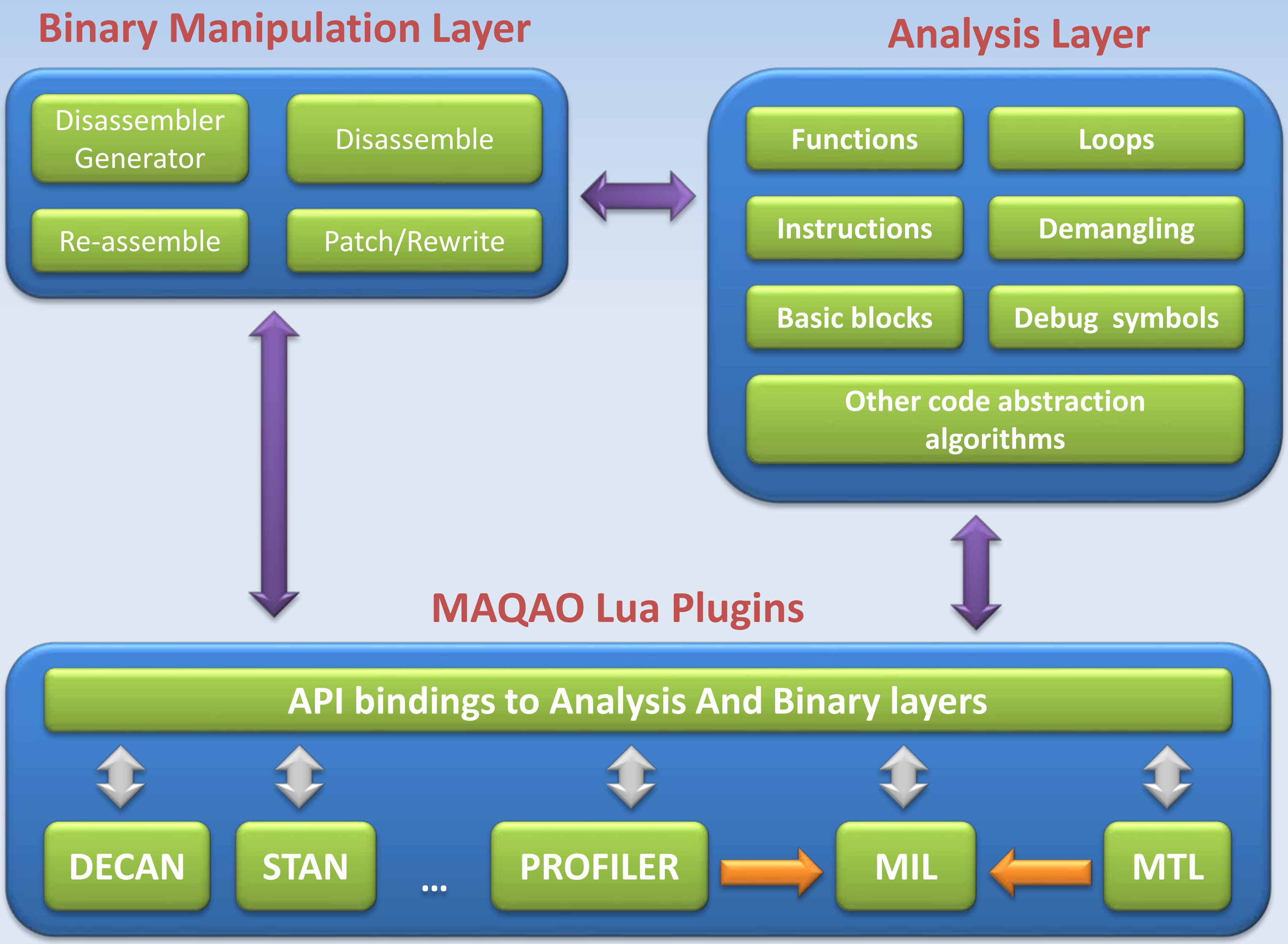
MAQAO Tool

Powerfull scripting interface

Example of script : Display memory instructions

```
--//Create a project and load a given binary
local project = project.new ("targeting load memomry instructions");
local bin = proj:load ( arg[1], 0);
--// Go through the abstract objects hierarchy and filter only load memory instructions
for f in bin:functions() do
  for l in f:innermost_loops() do
    for b in l:blocks() do
      for i in b:instructions() do
        if(i:is_load()) then
          local memory_operand = i:get_first_mem_oprnd();
          print(i);
          print(memory_operand);
        end
      end
    end
  end
end
```

MAQAO Framework



MAQAO Tool
Methodology

- Decision tree: smallest possible
- Detect hot spots:
  - Function (with/without callgraph) or loops (outer)
  - Include static estimation (sort functions)
- Code type characterization:
  - Through dynamic analysis (DECAN)
    - If memory bound: Memory behavior characterization
    - If compute bound: Static analysis
- Iterative approach:
  - user chooses to start over again if it is worth

MAQAO Tool
Contributions

- Contribute since 2006
- Old version, early days:
  - IA64: performance model, data dependency graph
  - Scripting interface (integration of Lua)
  - X86 assembly parser
- New version, during the thesis:
  - MIL
  - MTL
  - Profiler

Dynamic analysis

Outline

- Introduction
- Performance analysis
- Instrumentation Language
- Memory behavior characterization
- MAQAO tool
- Conclusion and Future work

Conclusion

- An instrumentation language to easily build custom performance evaluation tools
- A memory bahavior characterization tool
- A coase grain analysis tool: Profiler
- A methodology to analyze and optimize applications using MAQAO framework
- Contributions integrated into MAQAO tool along with external contributions

Future work

- Models: we studied OpenMP but not MPI
- Extend MIL:
  - More domain specific elements (counters,timers)
  - Complex events: support OpenMP
- Extend MTL:
  - Extend to OpenMP tasks
  - Integrate timing information: temporal aspects
  - Connect with runtimes
    - Get information (OpenMP: chunks size, strategy)
    - Provide information for betters decisions

Thanks for your attention !

Questions ?