Performance Evaluation
MAQAO Toolsuite

Ter@tec – 2nd July 2014
1. Introduction
2. PAMDA Methodology
3. MAQAO Framework
4. PerfEval: Profiling
5. CQA: Code Quality Analysis
6. DECAN: Differential Analysis
7. Success Stories
Introduction: Performance evaluation

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

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

- Several tools available
  - Which one to use?
  - \(\Rightarrow\) Need for a methodology
Introduction: Existing tools and methodologies (1/2)

- 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
Introduction: Motivating example

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 ; Ephem = Ephem + 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
```

Is it possible to:

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

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

~10% walltime

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

  - Profiling
  - Loops of interest
  - Differential analysis

  **CPU oriented**
  - Code Quality Analysis
  - Differential analysis
  - Value Profiling

  **Memory oriented**
  - Memory behavior characterization
  - Differential analysis
• 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
● Open source (LGPL 3.0)
  ● Currently binary release
  ● Source release soon

● Available for:
  ● x86-64
  ● Xeon Phi
MAQAO: Introduction

- **Audience**
  - User/Tool developer:
    - analysis and optimization tool
  - User/Tool developer:
  - 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*)

* MAQAO Instrumentation Language
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
- 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/….)
- 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Median Excl %Time</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>matmul_sub___ - <a href="mailto:56@solve_sub.f">56@solve_sub.f</a></td>
<td>17.16</td>
<td>0.26</td>
</tr>
<tr>
<td>compute_rhs___ - <a href="mailto:4@rhs.f">4@rhs.f</a></td>
<td>10</td>
<td>0.03</td>
</tr>
<tr>
<td>y_solve_cell___ - <a href="mailto:385@y_solve.f">385@y_solve.f</a></td>
<td>9.32</td>
<td>0.54</td>
</tr>
<tr>
<td>z_solve_cell___ - <a href="mailto:385@z_solve.f">385@z_solve.f</a></td>
<td>8.96</td>
<td>0.14</td>
</tr>
<tr>
<td>x_solve_cell___ - <a href="mailto:391@x_solve.f">391@x_solve.f</a></td>
<td>8.68</td>
<td>0.17</td>
</tr>
<tr>
<td>MPIDI_CH3I_Progress</td>
<td>5.22</td>
<td>3.66</td>
</tr>
<tr>
<td>matvec_sub___ - <a href="mailto:5@solve_sub.f">5@solve_sub.f</a></td>
<td>3.92</td>
<td>0.11</td>
</tr>
<tr>
<td>x_backsubstitute___ - <a href="mailto:330@x_solve.f">330@x_solve.f</a></td>
<td>3.09</td>
<td>0.14</td>
</tr>
<tr>
<td>y_backsubstitute___ - <a href="mailto:329@y_solve.f">329@y_solve.f</a></td>
<td>2.05</td>
<td>0.03</td>
</tr>
<tr>
<td>z_backsubstitute___ - <a href="mailto:329@z_solve.f">329@z_solve.f</a></td>
<td>1.98</td>
<td>0.06</td>
</tr>
<tr>
<td>copy_faces___ - <a href="mailto:4@copy_faces.f">4@copy_faces.f</a></td>
<td>0.88</td>
<td>0.06</td>
</tr>
<tr>
<td>MPID_nem_daplr_rc_poll_dyn_opt_</td>
<td>0.74</td>
<td>0.62</td>
</tr>
<tr>
<td>MPID_nem_limit_shm_start_send</td>
<td>0.68</td>
<td>0.06</td>
</tr>
</tbody>
</table>
(multi)node load balancing vue

Performance Evaluation - Profiling results

<table>
<thead>
<tr>
<th>Function</th>
<th>%Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>exact_solution__ - <a href="mailto:4@exact_solution.f">4@exact_solution.f</a></td>
<td>0.21</td>
</tr>
<tr>
<td>x_unpack_solve_info__ - <a href="mailto:114@x_solve.f">114@x_solve.f</a></td>
<td>0.14</td>
</tr>
</tbody>
</table>
# Node vue

## cirrus5003 - Process #53572 - Thread #1

<table>
<thead>
<tr>
<th>Name</th>
<th>Excl %Time</th>
<th>Excl Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>matmul_sub__ - <a href="mailto:56@solve_subsn.f">56@solve_subsn.f</a></td>
<td>16.92</td>
<td>16.48</td>
</tr>
<tr>
<td>▼ compute_rhs__ - <a href="mailto:4@rhs.f">4@rhs.f</a></td>
<td>9.92</td>
<td>9.66</td>
</tr>
<tr>
<td>▼ y_solve_cell__ - <a href="mailto:385@y_solve.f">385@y_solve.f</a></td>
<td>9.08</td>
<td>8.84</td>
</tr>
<tr>
<td>▼ loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▼ Loop 267 - y_solve.f@415</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 268 - y_solve.f@425</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 270 - y_solve.f@524</td>
<td>6.57</td>
<td></td>
</tr>
<tr>
<td>▼ x_solve_cell__ - <a href="mailto:391@x_solve.f">391@x_solve.f</a></td>
<td>9.01</td>
<td>8.78</td>
</tr>
<tr>
<td>▼ loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▼ Loop 235 - x_solve.f@420</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 236 - x_solve.f@429</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 237 - x_solve.f@709</td>
<td>0.06</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 238 - x_solve.f@519</td>
<td>6.24</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 239 - x_solve.f@431</td>
<td>2.71</td>
<td></td>
</tr>
</tbody>
</table>
### cirrus5003 - Process #53572 - Thread #1

<table>
<thead>
<tr>
<th>Name</th>
<th>Excl %Time</th>
<th>Excl Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>matmul_sub__ - 56@solve_subsf</td>
<td>16.92</td>
<td>16.48</td>
</tr>
<tr>
<td>▼ compute_rhs__ - <a href="mailto:4@rhs.f">4@rhs.f</a></td>
<td>9.92</td>
<td>9.66</td>
</tr>
<tr>
<td>▼ y_solve_cell__ - <a href="mailto:385@y_solve.f">385@y_solve.f</a></td>
<td>9.08</td>
<td>8.84</td>
</tr>
<tr>
<td>▼ loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▼ Loop 267 - y_solve.f@415</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 268 - y_solve.f@425</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 269 - y_solve.f@425</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 270 - y_solve.f@524</td>
<td>6.57</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 271 - y_solve.f@436</td>
<td>2.22</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 269 - y_solve.f@716</td>
<td>0.04</td>
<td></td>
</tr>
<tr>
<td>▼ x_solve_cell__ - <a href="mailto:391@x_solve.f">391@x_solve.f</a></td>
<td>9.01</td>
<td>8.78</td>
</tr>
<tr>
<td>▼ loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▼ Loop 235 - x_solve.f@420</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 236 - x_solve.f@429</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 237 - x_solve.f@709</td>
<td>0.06</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 239 - x_solve.f@431</td>
<td>2.71</td>
<td></td>
</tr>
<tr>
<td>▼ Loop 238 - x_solve.f@519</td>
<td>6.24</td>
<td></td>
</tr>
</tbody>
</table>
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

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))

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).
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
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
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

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
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 1500 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
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
Monitor:

- Execution times
- Loop Iteration numbers
- Hardware counter values
**DECAN: Polaris example**

- **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

Variable number of iterations
Non-unit stride accesses

```c
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+rwvalc1(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
ul1g = ul1*gE ; u2g = u2*gE ; u3g = u3*gE
g1c = g1c - ul1g ; g2c = g2c - u2g ; g3c = g3c - u3g
gr(nj1,thread_num) = gr(nj1,thread_num) + ul1g
gr(nj2,thread_num) = gr(nj2,thread_num) + u2g
gr(nj3,thread_num) = gr(nj3,thread_num) + u3g
```

High number of statements
Vector versus scalar

Non-unit stride accesses
Indirect accesses
Reducions
DIV/SQRT

- FP / LS transformations

ROI = FP / LS = 4.1

Imbalance between the two streams

=> Try to consume more elements inside one iteration.
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

**Execution time**

<table>
<thead>
<tr>
<th>Variants</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best_estimated</td>
<td></td>
</tr>
<tr>
<td>REF</td>
<td></td>
</tr>
<tr>
<td>FP</td>
<td></td>
</tr>
<tr>
<td>LS</td>
<td>5</td>
</tr>
<tr>
<td>REF_NSD</td>
<td>10</td>
</tr>
<tr>
<td>FPIS_NSD</td>
<td></td>
</tr>
</tbody>
</table>

**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
Acknowledgements

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

<table>
<thead>
<tr>
<th></th>
<th>Dyninst</th>
<th>PIN</th>
<th>PEBIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language type</td>
<td>API Oriented / DSL</td>
<td>API Oriented</td>
<td>API Oriented</td>
</tr>
<tr>
<td>Instrumentation type</td>
<td>Static/Dynamic binary</td>
<td>Dynamic binary</td>
<td>Static binary</td>
</tr>
<tr>
<td>Overhead</td>
<td>High/High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Safe Method</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

- **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.)
Example 1: TAU Profiler

```java
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);
```
Example 2: Filtering

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

Previous example only needs an additional statement