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

**BEFORE**

1 core at high frequency

**AFTER**

Multiple cores at lower frequency

Shift around 2005
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
Modern machines are very complex:

- Complex architectures: not easy to fully exploit
- Access to memory = huge impact: the memory wall
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
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: aggregated statistics ➔ tracing, sampling
### Performance Analysis

#### Existing tools

<table>
<thead>
<tr>
<th>Classification</th>
<th>Analysis Method</th>
<th>Measure</th>
<th>Insertion Level</th>
<th>User feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tools</strong></td>
<td>Modeling</td>
<td>Simulation</td>
<td>Tracing</td>
<td>Profiling</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gprof</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Callgrind</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP$IM$</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VTune</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAU</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HPCToolkit</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open</td>
<td>SpeedShop</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Scalasca</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SvPablo</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PerfExpert</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAQAO</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Andrés S CHARIF-RUBIAL

Ph.D. Defense - 22 October 2012
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

- Helpful analyses for users:
  - Provide useful 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

<table>
<thead>
<tr>
<th></th>
<th>Dyninst</th>
<th>PIN</th>
<th>PEBIL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language type</strong></td>
<td>API Oriented / DSL</td>
<td>API Oriented</td>
<td>API Oriented</td>
</tr>
<tr>
<td><strong>Instrumentation type</strong></td>
<td>Static/Dynamic binary</td>
<td>Dynamic binary</td>
<td>Static binary</td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td>High/High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Robust</strong></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
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 block relocation

Diagram:
- Entry
- Block 1
- Block 2
- Block 3
- Exit
- Probe

Content of basic block 3:
- `mov %edi,%r13d`
- `mov %rsi,%r14`
- `sar $0x3,%rbp`
- `mov %rdx,%r15`
- `callq 400f08 <Callee>`

End of binary:
- `callq 400fa8 <UserFn>`
- `callq 400f08 <Callee>`
- `jmp next instruction`

Relocation:
- `callq 400fa8 <UserFn>`
- `callq 400f08 <Callee>`
- `jmp next instruction`
Problem: instrumenting small basic blocks

<table>
<thead>
<tr>
<th>Method 1: using trampolines</th>
<th>Method 2: function relocation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function DoExec</strong></td>
<td><strong>Function DoExec</strong></td>
</tr>
<tr>
<td>Entry</td>
<td>Entry</td>
</tr>
<tr>
<td>Block1</td>
<td>Block1</td>
</tr>
<tr>
<td>Exit</td>
<td>Exit</td>
</tr>
<tr>
<td><strong>Probe</strong></td>
<td><strong>Probe</strong></td>
</tr>
</tbody>
</table>

Cannot deal with 1-Byte basic blocks

Cannot handle pointer (indirect branches)
Problem: instrumenting 1-Byte blocks

<table>
<thead>
<tr>
<th>Method 1: only OS signal handlers</th>
<th>Our method: use predecessors (CFG)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function DoExec</strong></td>
<td><strong>Function DoExec</strong></td>
</tr>
<tr>
<td>Entry</td>
<td>Entry</td>
</tr>
<tr>
<td>Block1</td>
<td>Block1</td>
</tr>
<tr>
<td>Block2</td>
<td>Block2</td>
</tr>
<tr>
<td>Block3</td>
<td>Block3</td>
</tr>
<tr>
<td>Exit</td>
<td>Exit</td>
</tr>
<tr>
<td>OS Signal</td>
<td>OS Signal</td>
</tr>
<tr>
<td>Probe</td>
<td>Probe</td>
</tr>
<tr>
<td><strong>Huge overhead</strong></td>
<td>Minimizes/Removes OS Signal execution</td>
</tr>
</tbody>
</table>

Exemple: dc.A (NPB-OMP) => 8x improvement
Resolve indirect jumps: locate hidden exits

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

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

Instrumentation Language
Advanced static analysis

Conditional Exit probe

%r14 < F.Start or %r14 > F.Stop

else

if

Exit Probe
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)
### Events: Where?

<table>
<thead>
<tr>
<th>Level</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Entry / Exit (avoid LD + exit handlers)</td>
</tr>
<tr>
<td>Function</td>
<td>Entries / Exits</td>
</tr>
<tr>
<td>Loop</td>
<td>Entries / Exits / Backedges</td>
</tr>
<tr>
<td>Block</td>
<td>Entry / Exit</td>
</tr>
<tr>
<td>Instruction</td>
<td>Before / After</td>
</tr>
<tr>
<td>Callsite</td>
<td>Before / After</td>
</tr>
</tbody>
</table>
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)
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)
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
Ex: TAU Profiler

```
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"} }
    }},
  }}
};
```
Integrated into TAU toolkit (previous example)

- tau_rewrite
- More expressive:
  - MIL: 20 lines
  - Dyninst: 200 lines

Ongoing integration with Score-P (H4H project)
- Using TAU profiler
- NPB-OMP: 12 threads
- More robust: all
- Faster: up to 8x
- JIT version (MILRT) remains affordable

1-Byte basic block problem

trampoline mechanism overhead

8x

4.5x

8x

Instrumentation Language

Comparing MIL and Dyninst overhead using TAU
### Instrumentation Language

#### Comparing MIL and Dyninst overhead using TAU

**Accuracy of results: output of thread1 for bt.A**

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive Name usec/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>1,164</td>
<td>1:07.796</td>
<td>1</td>
<td>1012</td>
<td>67796835 .TAU application</td>
</tr>
<tr>
<td>31.9</td>
<td>21,146</td>
<td>21,649</td>
<td>201</td>
<td>174096</td>
<td>107709 x_solve_omp</td>
</tr>
<tr>
<td>31.8</td>
<td>21,400</td>
<td>21,569</td>
<td>201</td>
<td>122492</td>
<td>107309 y_solve_omp</td>
</tr>
<tr>
<td>31.7</td>
<td>21,359</td>
<td>21,496</td>
<td>201</td>
<td>103416</td>
<td>106948 z_solve_omp</td>
</tr>
<tr>
<td>2.4</td>
<td>1,649</td>
<td>1,649</td>
<td>202</td>
<td>0</td>
<td>8167 compute_rhs_</td>
</tr>
<tr>
<td>0.2</td>
<td>156</td>
<td>156</td>
<td>201</td>
<td>0</td>
<td>776 add_</td>
</tr>
</tbody>
</table>

### Dyninst

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

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

- 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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Naive Overhead</th>
<th>Enhanced Overhead</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOMP 312.swim_m</td>
<td>273x</td>
<td>0.04x</td>
<td>6825x</td>
</tr>
<tr>
<td>SOMP 314.mgrid_m</td>
<td>974x</td>
<td>8.36x</td>
<td>116.5x</td>
</tr>
<tr>
<td>NAS PB ft.B</td>
<td>2160x</td>
<td>349x</td>
<td>6x</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Loop (MAQAO id)</th>
<th>% of Wall time</th>
</tr>
</thead>
<tbody>
<tr>
<td>flux_numerique_z</td>
<td>193</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>195</td>
<td></td>
</tr>
<tr>
<td>flux_numerique_x</td>
<td>204</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>206</td>
<td></td>
</tr>
</tbody>
</table>

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

Single threaded aspects: Inefficient patterns

Real code example: PNBENCH

```c
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

```c
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

```c
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 threads is not always the best choice
  - Find out the best thread number

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Reference</th>
<th>Best</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WTime (s)</td>
<td>Threads</td>
<td>WTime (s)</td>
</tr>
<tr>
<td>NPB CG.A</td>
<td>0.62</td>
<td>96</td>
<td>0.42</td>
</tr>
<tr>
<td>NPB FT.A</td>
<td>2.29</td>
<td>96</td>
<td>1.47</td>
</tr>
<tr>
<td>SOMP 320.mgrid_m R</td>
<td>111.14</td>
<td>40</td>
<td>84.71</td>
</tr>
<tr>
<td>SOMP 312.swim_m R</td>
<td>122.63</td>
<td>40</td>
<td>79.22</td>
</tr>
</tbody>
</table>
Memory behavior characterization
Understanding interactions between threads
Load balancing

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

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

Understanding interactions between threads

Data sharing

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

```javascript
--/Create a project and load a given binary
local project = project.new("targeting load memory 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 Tool
MAQAO Framework

Binary Manipulation Layer
- Disassembler Generator
- Re-assemble
- Disassemble
- Patch/Rewrite

Analysis Layer
- Functions
- Loops
- Instructions
- Demangling
- Basic blocks
- Debug symbols
- Other code abstraction algorithms

MAQAO Lua Plugins

API bindings to Analysis And Binary layers
- DECAN
- STAN
- ... PROFILER
- MIL
- MTL
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 behavior characterization tool
- A coarse 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 better decisions
Thanks for your attention!

Questions?