Static and Dynamic Approach for Performance Evaluation of Scientific Codes

Souad Koliaï

PhD advisor: William Jalby

University of Versailles Saint-Quentin-en-Yvelines, France
Exascale Computing Research Center, France
LRC ITACA, France

July 11th, 2012
1 Introduction

2 Static Analysis: MAQAO

3 Decremental Analysis: DECAN

4 Performance Evaluation Process

5 Conclusion
Computer Evolution & Bottleneck Detection

Optimization process: a *non-one step process*

- Characterize the code
- Diagnose the cause of the poor performance
- Prescribe a suitable optimization
## Optimization process: a non-one step process

- Characterize the code
- Diagnose the cause of the poor performance
- Prescribe a suitable optimization

## What to do to diagnose the problem?

Gather data about:
- Program
- Architecture
- Interaction between the two
Computer Evolution & Bottleneck Detection

Bottleneck detection: a tedious problem

- Hardware issues:
  - Microarchitecture
  - Memory

- Software issues:
  - Program behavior
  - Compiler

Performance Evaluation Tools: Automatic Bottleneck Detection
What makes the microarchitecture complex?

- Superscalar CPUs
- Pipelined CPUs
- Complex mechanisms:
  - Instruction pairing
  - Instruction fetching and decoding
  - Register renaming
  - Out of order execution
What makes the microarchitecture complex?

- Superscalar CPUs
- Pipelined CPUs
- Complex mechanisms:
  - Instruction pairing
  - Instruction fetching and decoding
  - Register renaming
  - Out of order execution

How to tackle this complexity?

- Performance modeling to mimic the pipeline behavior
Bottleneck Detection: Hardware Issues

Memory Wall

- Increasing gap between CPU & memory bus frequency
- CPU frequency = 2*Memory bus frequency
- Caches:
  - Associativity
  - Cache coherency
  - Prefetching (HW/SW)
Compiler Dependency

- Performance analysis on source code ⇒ simple
- From source to binary ⇒ optimizations
Bottleneck Detection: Software Issues

**Compiler Dependency**
- Performance analysis on source code ⇒ simple
- From source to binary ⇒ optimizations

**Compiler Independence**
- Compiler optimizations ⇒ non-controlled tuning
- Performance analysis on binary code ⇒ compiler-independent
How to tackle the bottleneck detection problem?

- Systematic/Automatic approach
- Different techniques, different strengths
Contributions

How to tackle the bottleneck detection problem?

- Systematic/Automatic approach
- Different techniques, different strengths

Thesis contribution

- Static and dynamic approach for a better evaluation process
Contributions

How to tackle the bottleneck detection problem?

- Systematic/Automatic approach
- Different techniques, different strengths

Thesis contribution

- Static and dynamic approach for a better evaluation process
- Static analysis: MAQAO
  - Tackles microarchitecture and compiler impact
Contributions

How to tackle the bottleneck detection problem?

- Systematic/Automatic approach
- Different techniques, different strengths

Thesis contribution

- Static and dynamic approach for a better evaluation process
- Static analysis: MAQAO
  - Tackles microarchitecture and compiler impact
- Dynamic analysis: DECAN
  - Tackles memory wall
1. Introduction

2. Static Analysis: MQAO

3. Decremental Analysis: DECAN

4. Performance Evaluation Process

5. Conclusion
Static Analysis in Performance Evaluation

**Static Analysis: Why?**

- First step in quality-control process
- Fast, abstracts dynamic phenomena
- Can be applied earlier in development
- Input dataset independent
- Detailed hints on code structure
Static Analysis in Performance Evaluation

Static Analysis: Why?
- First step in quality-control process
- Fast, abstracts dynamic phenomena
- Can be applied earlier in development
- Input dataset independent
- Detailed hints on code structure

Static Analysis: How?
- Performance modeling of microarchitecture
Motivating Example

DO cb=1,ncbt
  igp = isg;   isg = icolb(icb+1);   igt = isg + igp
  DO ig=1,igt
    e = ig + igp
    i = nnbar(e,1)
    j = nnbar(e,2)
    DO k=1,ndof
      DO l=1,ndof
        vecty(i,k) = vecty(i,k) + ompu(e,k,l)*vecx(j,l)
        vecty(j,k) = vecty(j,k) + ompl(e,k,l)*vecx(i,l)
      ENDDO
    ENDDO
  ENDDO
ENDDO
ENDDO
ENDDO
ENDDO

Sparse Matrix-Vector Product
Motivating Example

Execution ports in the Core 2 microarchitecture
Motivating Example

Execution ports in the Core 2 microarchitecture
Motivating Example

Execution ports in the Core 2 microarchitecture

- P0
- P1
- P2
- P3
- P4
- P5

FP Multiply
FP Addition
BRANCH
Motivating Example

Execution ports in the Core 2 microarchitecture
Motivating Example

Execution ports in the Core 2 microarchitecture
Motivating Example

Execution Ports

Dispatch on execution ports
Static Analysis in Performance Evaluation

Performance Modeling

- Mimics the microarchitecture behavior
- Focus on significant parts of the microarchitecture
- Gives performance predictions
- Detect inefficiencies statically
- No overhead
Static Analysis in Performance Evaluation

Static Analysis & Accuracy

- No knowledge about dynamic phenomena:
  - Data caching
  - Iteration count
Static Analysis in Performance Evaluation

Static Analysis & Accuracy

- No knowledge about dynamic phenomena:
  - Data caching
  - Iteration count

- Performance model:
  - Goes beyond than GFLOPS estimate
  - Gives detailed info on the microarchitecture behavior
Static Analysis in Performance Evaluation

Static Analysis & Accuracy

- No knowledge about dynamic phenomena:
  - Data caching
  - Iteration count

- Performance model:
  - Goes beyond than GFLOPS estimate
  - Gives detailed info on the microarchitecture behavior

- Core 2 and NHM plugins in M\textsc{aqao}
Static Analysis in MAQAO

MAQAO framework

- MAQAO: a Modular Assembly Quality Analyzer and Optimizer
- First version of MAQAO for the Itanium architecture
- Current version of MAQAO for the x86 architecture
Introduction

Static Analysis: **MAQAO**

Decremental Analysis: **DECAN**

Performance Evaluation Process

Conclusion

---

**Static Analysis in MAQAO**

**MAQAO framework**

- **MAQAO**: a Modular Assembly Quality Analyzer and Optimizer
- First version of **MAQAO** for the Itanium architecture
- Current version of **MAQAO** for the x86 architecture

- **MAQAO** = code restructuring + LUA plugins
- **Static analysis** performance model plugin for:
  - The Core2 architecture
  - NHM architecture
- Performance model in LUA for rapid prototyping
MAQAO Workflow from Binary to Code Quality Report
MAQAO Workflow from Binary to Code Quality Report

Binary → Disassembling → Code Quality Report
MAQAO Workflow from Binary to Code Quality Report

Binary → Disassembling → Code Restructuring

- Call Graph
  - Control Flow Graph
  - Data Dependence Graph
  - Loop Detection

→ Code Quality Report
**MAQAO Workflow from Binary to Code Quality Report**

- **Binary**
- **Disassembling**
- **Code Restructuring**
  - Call Graph
    - Control Flow Graph
    - Data Dependence Graph
    - Loop Detection
- **LUA Plugin 1**
- **LUA Plugin N**
- **Code Quality Report**
MAQAO Workflow from Binary to Code Quality Report

Binary → Disassembling → Code Restructuring

- Call Graph
  - Control Flow Graph
  - Data Dependence Graph
  - Loop Detection

- LUA Plugin 1
- LUA Plugin i:
  - Performance Model
    - On inner loops

- LUA Plugin N

Code Quality Report
Static Analysis in MAQAO

Performance Model in MAQAO

- Computes asymptotic estimation
- Evaluates inner loop execution time
- Simulates the front-end and the back-end of the Core 2/NHM pipelines
Performance Modeling
Performance Modeling

Instruction Fetch Unit (IFU)
Max 1 16-byte block per cycle

Front-End

Front-End

Back-End
Performance Modeling

Instruction Fetch Unit (IFU) -> Predecoder -> Instruction Queue (IQ)

Max 1 16-byte block per cycle
Max 6 instructions per cycle

Front-End

Back-End
Performance Modeling

Front-End

Instruction Fetch Unit (IFU) → Predecoder → Instruction Queue (IQ) → Decoders → Decode Queue (IDQ)

Max 1 16-byte block per cycle
Max 6 instructions per cycle
Max 4 instructions per cycle
Max 7 uops per cycle

Front-End

Back-End
Performance Modeling
Performance Modeling

Diagram:
- Instruction Fetch Unit (IFU) to Predecoder: Max 1 16-byte block per cycle
- Predecoder to Instruction Queue (IQ): Max 6 instructions per cycle
- Instruction Queue (IQ) to Decoders: Max 4 instructions per cycle
- Decoders to Instruction Decode Queue (IDQ): Max 7 uops per cycle

Front-End:
- Instruction Decode Queue (IDQ) to RAT / ROB read: Max 4 uops per cycle
- RAT / ROB read to Dispatcher: Max 3 registers reads per cycle
- Dispatcher to Back-End: Max 6 uops per cycle

Back-End:
- Dispatcher to Instruction Decode Queue (IDQ)

Note:
- Instruction Fetch Unit (IFU) to Predecoder
- Predecoder to Instruction Queue (IQ)
- Instruction Queue (IQ) to Decoders
- Decoders to Instruction Decode Queue (IDQ)
- Instruction Decode Queue (IDQ) to RAT / ROB read
- RAT / ROB read to Dispatcher
- Dispatcher to Back-End

Max values:
- Max 1 16-byte block per cycle
- Max 6 instructions per cycle
- Max 4 instructions per cycle
- Max 7 uops per cycle
- Max 4 uops per cycle
- Max 3 registers reads per cycle
- Max 6 uops per cycle
Performance Modeling

Front-End

Instruction Fetch Unit (IFU) -> **Predecoder** -> Instruction Queue (IQ) -> **Decoders** -> Instruction Decode Queue (IDQ)

Max 1 16-byte block per cycle
Max 6 instructions per cycle
Max 4 instructions per cycle
Max 6 uops per cycle
Max 7 uops per cycle
Max 6 uops per cycle
Max 3 registers reads per cycle
Max 6 uops per cycle
Max 3 registers reads per cycle

RAT / ROB read

Dispatcher -> **Execution units** -> Retirement Unit

Max 4 uops per cycle
Max 3 registers reads per cycle
Max 6 uops per cycle
Max 6 uops per cycle
Max 6 uops per cycle

Back-End


Performance Modeling

Front-End
- Predecoder
  - Max 1 16-byte block per cycle
- Decoders
  - Max 4 instructions per cycle
  - Max 7 uops per cycle

RAT / ROB read
- Max 4 uops per cycle
  - Max 3 registers reads per cycle

Dispatcher
- Max 6 uops per cycle

Execution units
- Max 6 uops per cycle

Back-End
Performance Modeling

Vectorization

- Instruction level parallelization
- Operations on vector operands
- 128,256-bit vector in modern processors
# Performance Modeling

## Vectorization
- Instruction level parallelization
- Operations on vector operands
- 128,256-bit vector in modern processors

## Vectorization & Performance Model
- Evaluates the compiler vectorizing capabilities
- A ratio of 1 means that the code is fully vectorized
- A ratio of 0 means that the code is not vectorized
Performance Modeling

for (i=0 ; i<N ; i++)
    y[i] += alpha*x[i]

B1: Non vectorized

```
 movsd (%rdi,%rax,8),%xmm1
 mulsd %xmm0, %xmm1
 addsd (%rsi,%rax,8), %xmm1
 movsd %xmm1, (%rsi,%rax,8)
 incq %rax
 cmpq %r8, %rax
 jb B1
```
Performance Modeling

```c
for (i=0 ; i<N ; i++)
    y[i] += alpha*x[i]
```

B1: **Non vectorized**

```assembly
movsd (%rdi,%rax,8),%xmm1
mulsd %xmm0, %xmm1
addsd (%rsi,%rax,8), %xmm1
movsd %xmm1, (%rsi,%rax,8)
incq %rax
cmpq %r8, %rax
jb B1
```

```
  | X   | 1   | 2   | ... | i   | i+1 | ... | N-1 | N   |
  | Y   | 1   | 2   | ... | i   | i+1 | ... | N-1 | N   |
```
Performance Modeling

B1: Vectorized
movaps (%rdi,%rax,8),%xmm1
mulpd %xmm0, %xmm1
addpd (%rsi,%rax,8), %xmm1
movaps %xmm1, (%rsi,%rax,8)
addq $16, %rax
cmpq %r8, %rax
jb B1

for (i=0 ; i<N ; i++)
y[i] += alpha*x[i]

X
1 2 ... i i+1 ... N-1 N

Y
1 2 ... i i+1 ... N-1 N
Performance Modeling

for (i=0 ; i<N ; i++)
    y[i] += alpha*x[i]

B1: Vectorized
 movaps (%rdi,%rax,8),%xmm1
 mulpd %xmm0, %xmm1
 addpd (%rsi,%rax,8), %xmm1
 movaps %xmm1, (%rsi,%rax,8)
 addq $16, %rax
 cmpq %r8, %rax
 jb B1

\[
\begin{array}{cccccc}
X & 1 & 2 & \ldots & i & i+1 & \ldots & N-1 & N \\
\hline
Y & 1 & 2 & \ldots & i & i+1 & \ldots & N-1 & N \\
\end{array}
\]
Performance Modeling

Code Characteristics & Performance Model

- Bytes loaded/stored per cycle: memory traffic
- Vector registers used: register spilling
Performance Modeling

Code Characteristics & Performance Model
- Bytes loaded/stored per cycle: memory traffic
- Vector registers used: register spilling

Performance Prediction & Performance Model
- Performance prediction for:
  - Vectorization
  - L1 = max(front-end, back-end)
  - L2/RAM: pattern matching
Performance Modeling

Performance Prediction & Pattern Matching

- A step further in static analysis
- A bridge from static to dynamic analysis

Target Code

```assembly
Movaps (%rax),%xmm0
Movaps (%rcx),%xmm1
...
Movaps (%r10),%xmm3
Movaps %xmm4,(%rdx)
Movaps (%rbx),%xmm9
...
Movaps (%r15),%xmm0
```

Pattern Matching

Micro-Kernels Data-Base

Performance Prediction
1 Introduction

2 Static Analysis: MQAQA

3 Decremental Analysis: DECAN

4 Performance Evaluation Process

5 Conclusion
Dynamic Analysis in Performance Evaluation

Dynamic Analysis: Why?
- Complements the static analysis
- Detects dynamic dependencies
- Collects temporal information
- Deals with input dataset
Dynamic Analysis in Performance Evaluation

Dynamic Analysis: How?

- A new approach in dynamic analysis: Decan
- Deeper in program understanding
- Use static analysis report to drive the analysis
**DECAN: Concept and Workflow**

**DECAN concept**

- Fine-grained bottleneck detection approach
- Similar to the debug process:
  - A bug occurs
  - $\Rightarrow$ Code transformation
  - $\Rightarrow$ Run the code to check if the bug still occur or not
- Poor performance considered as a bug
### How Decan works?

- Performs on critical routines
- Targets inner loops
- Performs via binary patching
- Detects memory access impact
- Focuses on Streaming SIMD Extensions instructions (SSE) ⇒ vector instructions
DECAN: Concept and Workflow

Original Binary → Critical Routine

For Each Loop
Perform Instruction Patching

BinA → A
BinB → B
BinC → C
BinD → D
BinE → E

Execution in Original Context

A: one load / one store patched
B: all loads patched
C: all stores patched
D: all loads-stores patched
E: grouping
Motivating Example

4K-aliasing detection

- A false dependency between loads of $acx(i-1,j,k)$, $temp(i-1,j,k)$ and the store of $vhilf(i,j,k)$
- Serialization of the memory accesses
- False dependency due to 4K-aliasing

Matrix-Vector product loop

```
   do k = anf3, end3
     do j = anf2, end2
       do i = anf1, end1
         vhilf(i,j,k) = temp(i,j,k) - (  
          &   (acx(i-1,j,k) + acy(i,j-1,k) + acz(i,j,k) -  
          &   acx(i,j,k) + acy(i,j+1,k) + acz(i,j,k-1)) / coeffd(i,j,k)  
         )
         end do
       enddo
     enddo
   enddo
```
Motivating Example

What is 4K-aliasing?

Suppose the following portion of code:

```
for (i=0 , i<SIZE , i++)
a(i) = b(i-offset)
```

- If \((\text{add}(a) \mod 4\text{KB}) = (\text{add}(b) \mod 4\text{KB})\) (the same lower 12 bits)
- With offset = 1 there is a conflict between:
  - the store \(a(i)\) at iteration \(i\)
  - the load \(b((i+1)-1)\) at iteration \(i+1\)
Motivating Example

Impact of load/store instructions on Matvec subroutine

Transformed binaries generated by DECAN

Original

matvec_loop2_stores_loads
matvec_loop2_stores
matvec_loop2_st_0x403e5a
matvec_loop2_loads
matvec_loop2_ld_0x403e50
matvec_loop2_ld_0x403e36
matvec_loop2_ld_0x403e2c
matvec_loop2_ld_0x403e1c
matvec_loop2_ld_0x403d98
matvec_loop2_ld_0x403de1
matvec_loop2.ld_0x403dd3
matvec_loop2.ld_0x403d8
matvec_loop2.ld_0x403e12
matvec_loop2.ld_0x403e08
matvec_loop2.ld_0x403e1c
matvec_loop2.ld_0x403e26
matvec_loop2.ld_0x403e36
matvec_loop2.ld_0x403e50

Cycles
**Decan**: How to discard an instruction?

**Instruction Removal**
- From SSE memory instructions to *nop*
- Performed via binary patching
**DECAN**: How to discard an instruction?

**Instruction Removal**
- From SSE memory instructions to *nop*
- Performed via binary patching

For any loop containing *n* instructions, **DECAN** generates:
- *n* versions, each one corresponds to the removal of one memory access
- One version without any load
- One version without any store
- One version without any load/store

To avoid artificial pressure on the execution ports
To keep instruction alignment unchanged
**Decan**: How to discard an instruction?

**Instruction Removal**

- From SSE memory instructions to *nop*
- Performed via binary patching

- For any loop containing $n$ instructions, Decan generates:
  - $n$ versions, each one corresponds to the removal of one memory access
  - One version without any load
  - One version without any store
  - One version without any load/store

- The *nop* used to patch has the same size than the suppressed instruction:
  - To avoid artificial pressure on the execution ports
  - To keep instruction alignment unchanged
Illustrating Example

Consider the following vector addition:

```
for (i=0 ; i<size ; i+=2)  
  y[i] += alpha*x[i]
```

```
B1:  
  movsd (%rdi,%rax,8),%xmm1  
  mulsd %xmm0, %xmm1  
  addsd (%rsi,%rax,8), %xmm1  
  movsd %xmm1, (%rsi,%rax,8)  
  addq $16, %rax  
  cmpq %r8, %rax  
  jb B1
```
### Illustrating Example

The patching is performed as follows:

**B1:** #One load is patched  

<table>
<thead>
<tr>
<th><strong>nop</strong></th>
<th><strong>operand</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>mulsd</td>
<td>%xmm0, %xmm1</td>
</tr>
<tr>
<td>addsd</td>
<td>(%rsi,%rax,8), %xmm1</td>
</tr>
<tr>
<td>movsd</td>
<td>%xmm1, (%rsi,%rax,8)</td>
</tr>
<tr>
<td>addq</td>
<td>$16,%rax</td>
</tr>
<tr>
<td>cmpq</td>
<td>%r8, %rax</td>
</tr>
<tr>
<td>jb</td>
<td>B1</td>
</tr>
</tbody>
</table>
Illustrating Example

The patching is performed as follows:

B1: #One load is patched

```
nop
mulsd
addsd
movsd
addq
cmpq
jb
```

```
operand
%xmm0, %xmm1
(%rsi,%rax,8), %xmm1
%xmm1, (%rsi,%rax,8)
$16,%rax
%r8, %rax
B1
```

B1: #All loads are patched

```
nop
mulsd
nop
movsd
addq
cmpq
jb
```

```
operand
%xmm0, %xmm1
operand
%xmm1, (%rsi,%rax,8)
$16, %rax
%r8, %rax
B1
```
Decan: How to discard an instruction?

Grouping version of patching

- A group is a set of memory instructions that are using the same base address, i.e. accessing to the same array.
- Decan patches a group to detect the impact of these accesses on performance.
**DECAN**: How to discard an instruction?

```c
for (iif = 3, ic = 2 ; ic < nc ; ic++, iif += 2){
    uc[ic][jc] = 0.5 * uf[iif][jf] + 0.125 * (uc[iif + 1][jf] + uf[iif - 1][jf] + uf[iif][jf + 1] + uc[iif][jf - 1]);
}
```
**Decan**: How to discard an instruction?

```c
for (iif = 3, ic = 2 ; ic < nc ; ic++, iif += 2){
    uc[ic][jc] = 0.5 * uf[iif][jf] + 0.125 * (uc[iif + 1][jf] + uf[iif - 1][jf] + uf[iif][jf + 1] + uc[iif][jf - 1]);
}
```

```
..B1.4:

movsd   (%r8,%rdi), %xmm2
incq   %r11
movsd   (%r8,%r9), %xmm3
mulsd  %xmm1, %xmm3
addsd  (%r8,%rsi), %xmm2
addsd  8(%r8,%r9), %xmm2
addsd  -8(%r8,%rbx), %xmm2
mulsd  %xmm0, %xmm2
addsd  %xmm2, %xmm3
movsd  %xmm3, (%r15,%r12)
movsd  (%r8,%rax), %xmm4
movsd  (%r8,%rcx), %xmm5
mulsd  %xmm1, %xmm5
addsd  (%r8,%rdx), %xmm4
addsd  8(%r8,%rcx), %xmm
addsd  -8(%r8,%r14), %xmm4
mulsd  %xmm0, %xmm4
addq  -16(%rsp), %r8
addsd  %xmm4, %xmm5
movsd  %xmm5, (%r15,%r10)
addq  %rbp, %r15
cmpq  %r13, %r11
jb  ..B1.4
```
Decremental Analysis Limitations

- Dealing side effects when patching instructions
- Semantics are lost
- This is:
  - A fine-grained approach
  - Coupled with profiling
**Decan: Error Handling**

- **Decan** alters the semantics of the code.
- Is not considered in the analysis any binary leading to a crash.
DECAN: Error Handling

**Crash**
- **DECAN** alters the semantics of the code
- Is not considered in the analysis any binary leading to a crash

**FP Exception**
- Detected and counted in **DECAN** binaries
- Any binary that generates FPE is removed from the analysis
1. Introduction

2. Static Analysis: MQAO

3. Decremental Analysis: DECAN

4. Performance Evaluation Process

5. Conclusion
Toward a better evaluation process

Performance Evaluation Methodology: Why?

- Key factor for:
  - A good optimization
  - A bottleneck detection in the minimum time
- Systematic process $\Rightarrow$ good return on investment
- Not a theoretical concept
Toward a better evaluation process

Performance Evaluation Methodology: Why?

- Key factor for:
  - A good optimization
  - A bottleneck detection in the minimum time
- Systematic process $\Rightarrow$ good return on investment
- Not a theoretical concept

- Performance Evaluation Process is like a recipe
- Choose the best ingredients
- Choose ingredients that go well
Toward a better evaluation process

Performance Evaluation Methodology: How?

- Combine complementary ingredients/analyses:
  - Profiling to detect critical routine
  - Static analysis to estimate quality of the code
  - Dynamic analysis to pinpoint the delinquent memory access

- At each iteration of the process, we go deeper in the understanding of the program behavior
Toward a better evaluation process

<table>
<thead>
<tr>
<th>Tools and target features</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F1</strong> Vectorization</td>
</tr>
<tr>
<td><strong>F2</strong> Dispersal on execution ports</td>
</tr>
<tr>
<td><strong>F3</strong> Estimation bound in L1, L2, and RAM</td>
</tr>
<tr>
<td><strong>F4</strong> Cache misses</td>
</tr>
<tr>
<td><strong>F5</strong> Load-Store impact</td>
</tr>
<tr>
<td><strong>F6</strong> 4K-aliasing</td>
</tr>
<tr>
<td><strong>F7</strong> Memory access patterns.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maqao</strong></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Decan</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Hardware Performance Monitoring</strong></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Memory Traces</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Evaluation Process on Industrial Program

RECOM application

- Builds a 3D model of industrial-scale furnaces
- Critical routine: \textit{RBgauss}
- Implements a \textit{Red-Black} solver

```plaintext
DO IDO=1,NREDD
   INC = INDIRR(IDO)
   HANB = AM(INC,1)*PHI(INC+1) &
        + AM(INC,2)*PHI(INC-1) &
        + AM(INC,3)*PHI(INC+INPD) &
        + AM(INC,4)*PHI(INC-INPD) &
        + AM(INC,5)*PHI(INC+NIJ) &
        + AM(INC,6)*PHI(INC-NIJ) &
        + SU(INC)
   DLTPHI = UREL*( HANB/AM(INC,7) - PHI(INC) )
   PHI(INC) = PHI(INC) + DLTPHI
   RESI = RESI + ABS(DLTPHI)
   RSUM = RSUM + ABS(PHI(INC))
ENDDO
```

Most-time-consuming loop in \textit{RBgauss}
Static Analysis on RECOM

*Static Analysis* detects:
- Loop not vectorized
- Loop memory bound
- Lower bound that can be achieved
Static Analysis on RECOM

*Static Analysis* detects:
- Loop not vectorized
- Loop memory bound
- Lower bound that can be achieved
Decremental Analysis on RECOM - All patch versions

Impact of load/store instructions on RBgauss subroutine

Original vs. Transformed binaries generated by DECAN
Decremental Analysis on RECOM - Grouping version

Recom application - Grouping of SSE memory instructions that access to the same base address (AM array)
Optimization

- **Stride 2 access on AM array**
- **Split AM into two arrays with a stride 1 access**

```plaintext
DO IDO=1,NREDD
    INC = INDINR(IDO)

    HANB = AM(INC,1)*PHI(INC+1) &
           + AM(INC,2)*PHI(INC-1) &
           + AM(INC,3)*PHI(INC+INPD) &
           + AM(INC,4)*PHI(INC-INPD) &
           + AM(INC,5)*PHI(INC+NIJ) &
           + AM(INC,6)*PHI(INC-NIJ) &
           + SU(INC)

    DLTPHI = UREL*( HANB/AM(INC,7) - PHI(INC) )
    PHI(INC) = PHI(INC) + DLTPHI

    RESI = RESI + ABS(DLTPHI)
    RSUM = RSUM + ABS(PHI(INC))
ENDDO

Most-time-consuming loop in RBgauss

```

```plaintext
DO IDO=1,NREDD
    INC = INDINR(IDO)
    INC_AMR = INDAMR(IDO)

    HANB = AMR(INC_AMR,1)*PHI(INC+1) &
           + AMR(INC_AMR,2)*PHI(INC-1) &
           + AMR(INC_AMR,3)*PHI(INC+INPD) &
           + AMR(INC_AMR,4)*PHI(INC-INPD) &
           + AMR(INC_AMR,5)*PHI(INC+NIJ) &
           + AMR(INC_AMR,6)*PHI(INC-NIJ) &
           + SU(INC)

    DLTPHI = UREL*( HANB/AMR(INC_AMR,7) - PHI(INC) )
    PHI(INC) = PHI(INC) + DLTPHI

    RESI = RESI + ABS(DLTPHI)
    RSUM = RSUM + ABS(PHI(INC))
ENDDO

Most-time-consuming loop in RBgauss - Optimized
```
Speedup achieved in unicore in *RBgauss* subroutine: 1.3
Speedup achieved in multicore in RECOM application: 1.4

![AIOLOS application diagram](image-url)
Performance Evaluation Methodology

- A good performance evaluation for a good optimization
- Combine complementary techniques/strengths
- **MAQAO** for static analysis
- **DECAN**, HPM, Memory tracing for dynamic analysis
- Validated experimentally on industrial applications
- Speedup achieved on industrial applications
MAQAO Static Analysis

- Considered as a first step in a performance evaluation process
- Fast, abstracts the dynamic phenomena
- A performance model for the Core2 and NHM microarchitectures
- Will be extended to new x86 microarchitectures
**DECAN Decremental Analysis**

- **DECAN**, decremental analysis tool
- New approach for a fine-grained analysis
- Simple concept, similar to the debug process
- Compiler-independent, target binary codes
- Quantify dynamic phenomena
Future Work

- Extend MAQAO static analysis to new x86 microarchitectures: Sandy Bridge
Future Work

- Extend \texttt{M\textsc{aqao}} static analysis to new x86 microarchitectures: Sandy Bridge

- In \texttt{D\textsc{ecan}}, address loops with control flow
Future Work

- Extend MAQAO static analysis to new x86 microarchitectures: Sandy Bridge

- In DECAN, address loops with control flow
- Extend the concept of Decremental Analysis to go from instruction to threads’ tasks
Thank you!