A Decremental Analysis Tool for Fine-Grained Bottleneck Detection

Souad Koliaï¹,² Sébastien Valat¹,² Tipp Moseley³
Jean-Thomas Acquaviva¹,² William Jalby¹,²

¹University of Versailles Saint-Quentin-en-Yvelines, France
²Exatec-Lab, France
³Google, Mountain View, CA
Outline

• Introduction: my personal view on hardware performance counters
• DECAN: what?
• DECAN: how?
• Case studies
• Future work
How to deal with performance issues (1)

- First (well known) technique: profiling
  - Down to a few hot routines
  - Then analyze loop behavior
  - Four key issues: source code, compiler, OS, hardware

- Second analyze loops statically (source code, compiler)
  - Static analysis (MAQAO)
  - Allows to detect compiler inefficiencies
  - Provides performance estimates and bottleneck analysis

- In general discrepancy between static estimates and measurements
  - What is the next step ?
  - Use performance counters to get an idea of hardware performance behavior

A Decremental Analysis Tool for a Fine-Grained Bottleneck Detection
• Once you know the performance issues, analyze/evaluate them
  ▪ Sort them out by performance impact importance (ROI)
  ▪ Trade off between cost and potential performance gains

• After performance problem analysis, fix performance issues
  ▪ The main “performance knob” at our disposal are instructions
  ▪ Change the source code or assembly to remove performance issues

• Importance of ROI (Return On Investment)
  ▪ Routine A consumes 40% of execution time and performance gains are estimated on routine A at 10%: overall gain 4%
  ▪ Routine B consumes 20% of execution time and performance gains are estimated on routine B at 50%: overall gain 10%
In general, performance events give an aggregate view of the routine/loop behavior:

- Number of cache misses
- All of the instructions are “lumped” together: no individual view/report of an instruction
- REMEMBER: our main knob is at instruction level
Consider the C kernel:

```c
for (int i = 0 ; i < SIZE ; ++i )
a[ i ] = b[ i – offset ]
```

If we have addresses such as:

```
a % 4kB = b % 4kB (same low order 12 bits)
```

With `offset = 1`, there is a conflict between:
- The `store a[ (i) ]` from iteration `i`
- The `load b[ (i+1) - 1 ]` from iteration `i+1`

**THIS IS KNOWN AS THE 4 KB ALIASING PROBLEM**

This can be detected with hardware counter:
```
LOAD_BLOCK.OVERLAP_STORE
```
Performance on Intel CORE 2 duo

a[i] = b[i - offset] ; sizeof(a,b) = 512Ko
Core 2 Duo

A Decremental Analysis Tool for a Fine-Grained Bottleneck Detection
Sensible impact up to:

- **Offset = 10** in terms of counter
- **Offset = 4** in terms of time cost

The counter DETECTS the issue, but not the cost.

WHAT WE CARE ABOUT IS PERFORMANCE IMPACT
Hardware performance counters/events issues (1)

- Detects the source of the problem not the performance impact
  - Counts the number of 4 KB alias conflicts but not the cost
  - Counts the number of cache misses not the latency (except EAR on IA64 and mem lat counter on I7) and in fact you want the exposed latency 😊

- Sampling bias and threshold
  - Quantum measurement: every 100 000 cache misses, update counters
  - In general unable to assign the cost to the right/offending instruction
  - Delays between the counter overflow and the interrupt handler
  - Too many instructions in flight
  - Several instructions retiring at the same time
  - IN CONCLUSION BAD ACCOUNTING: NO GOOD CORRELATION WITH SOURCE CODE
Other Key issues with performance counters/events:
- TOO MANY OF PERFORMANCE EVENTS: Over 1200 on core I7
- TOO FEW COUNTERS: typically 4, getting values for all events would require 400 runs
- Deals with low level hardware and gives you a fragmented picture: counts the number of times prefetch are launched including the aborted cases
- Documentation is usually poor
- Needs to know very well micro architecture and in general corresponding info is not available
- Not consistent across architectures even on successive X86 generations

An interesting OLD idea: Profile me (DEC)
- Sample instructions
- Reports all stalls occurring to an instruction
• Be a physicist:
  ▪ Consider the machine as a black box
  ▪ Send signals in: code fragments
  ▪ Observe/measure signals out: time and maybe other metrics

• Signals in/Signals out
  ▪ Slightly modify incoming signals and observe difference/variations in signals out
  ▪ Tight control on incoming signal

• In coming signal: code
  ▪ Modify source code: easy but dangerous: the compiler is in the way
  ▪ Modify assembly/binary: much finer control but cautious about correlation with source code
• GOAL 1: detect the offending/delinquent operations

• GOAL 2: get an idea of potential performance gain
DECAN: What?

- A tool for fine grained detection of the bottleneck (ie. assembly instruction level)

- Focus on the hottest region of an application using automatic kernel extraction (AKE)

- DECAN performs on a binary and on loop level

- DECAN uses MAQAO/MADRAS disassembler tool chain
DECAN: General Concept

- DECAN’s concept is simple:
  - Measure the original binary
  - Patch the memory access instructions in the original binary
  - New binary is generated for each patch
  - Measure new binaries
  - Measurements are represented in a CSV file: analysis and comparison
Strategy for performance measurements: Automatic driver to extract a kernel from a given application

Goal:

- Focus on only a small part of the application (the hottest subroutine = the kernel)
- Extract the kernel and its memory context
- Build a driver to run the kernel in its original execution environment
Kernel extraction methodology

- Dump the memory context of the kernel using GDB
- Dump the parameters addresses of the kernel using GDB
- Map the memory context dumped
- Pass the parameters addresses dumped to the correct registers/stack location → generates a caller to the kernel
- Original memory context + correct calling convention → operational loader
- Bypass the main of the original application to branch to the loader → run the kernel in its original execution environment
DECAN focuses on SSE memory access instructions (ie. SSE loads and stores)

Memory access instruction patching:
- Replace the memory access instruction by a nop operation or a pxor to avoid extra dependencies
- Example:
  
  \[
  \text{movaps}\ (%rsi),\%xmm1 \Rightarrow \text{nop } r/m \text{ or pxor } \%xmm1, \%xmm1
  \]
  
  \[
  \text{movaps } \%xmm2, (%rsi) \Rightarrow \text{nop } r/m
  \]

Each patched instruction generates a new binary
DECAN: Instruction Patching

• If n SSE instructions then n+3 different binaries + grouping version of binaries are generated:
  - One_Load binary
  - One_Store binary
  - All_Loads binary
  - All_Stores binary
  - All_Loads_Stores binary
  - Grouping

• Each new binary has the following file name format:
  
  `<func_name> loopID OPT

OPT = loads|stores|loads_stores|(ld|st) @inst lineSRC

Example:
  
  `rbgauss_loop3 ld 0x402f4c line97` → in loop 3 of rbgauss function, the load instruction at 0x402f4c address has been modified (source line: 97)
• The original binary and the new binaries are measured using the automatic kernel extraction

• Performance measurements are gathered in a CSV file

• The CSV format allows to make easy the comparison between the original binary and the modified binaries and to pinpoint the delinquent memory access instruction
MAGMA is an application for the simulation of casting processes.

The hottest subroutine in MAGMA application is CGSolv.

The target loop in CGSolv is Matvec shown in Fig. 3.

Applying DECAN on Matvec generates a set of binaries (when modifying memory access instructions).

Performance measurements are gathered in MATVEC.csv file.

```fortran
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) * temp(i-1,j,k) )
        & + acx(i,j,k) * temp(i+1,j,k) )
        & + acy(i,j-1,k) * temp(i,j-1,k) )
        & + acy(i,j,k) * temp(i,j+1,k) )
        & + acz(i,j,k-1) * temp(i,j,k-1) )
        & + acz(i,j,k) * temp(i,j,k+1) )
      / coeffd(i,j,k)
    end do
  end do
end do
```

Fig. 3. Target Loop in CGSolv
• When replacing one load at the same time, there is some performance impact of the replaced load: however some loads have a larger impact than others.

• When replacing all loads, performance is improved by a factor of 2.5.

• When replacing a single store, performance is improved by a factor of 2.5 → this store seems to be the bottleneck.

• Conclusion: the conflict between the loads and a store seems to be the bottleneck!

• A 4K-aliasing load-store conflict between vhlf (the array being stored), temp and acx (the arrays being loaded).
• RECOM application builds a 3D-model of industrial-scale furnaces.

• The hottest subroutine in Recom application is RBgauss

• The target loop in RBgauss is shown in Fig.1

• 3D structures (arrays, loops) are linearized

• Regular geometry but with holes: use of indirect access to jump over holes

• RB stands for Red Black: many access are stride 2

```
DO IDO=1,NREDD
   INC = INDFTR(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
```

*Fig. 1. Target Loop in RBgauss*
Impact of load/store instructions on RBgauss subroutine

Transformed binaries generated by DECAN
DECAN refinement: instruction grouping

- An example:
  - $B(i) = A(I) + A(I+1)$
  - Let us assume $A$ coming from memory: 1 miss followed by a hit
  - Nopping $A(I)$ generates one miss $A(I+1)$
  - Nopping $A(I+1)$ generates one miss on $A(I)$

- Basic idea of grouping
  - Group together loads which are dependant upon each other
  - Group loads accessing the same array

- How to implement grouping
  - Analyze start array address
  - Group together loads which corresponds to “close” start array address
Recom application - Grouping of SSE memory instructions that access to the same base address (AM array)
• When nooping one load at the same time, there is limited effect of the nopped load.

• When replacing all loads, performance is improved by a factor of 1.75

• Grouping shows that most of the performance loss is associated with **access to a 1D array** : AM

• Conclusion: AM access seems to be the bottleneck!

• A memory trace tool is used to detect how AM is accessed

• AM is accessed with a STRIDE 2!: solution: restructure splitting AM into two distinct arrays: one for the RED, one for the BLACK
DECAN: Case Studies - RECOM

- Limiting array restructuring to AM is much simpler: read only structure
- Restructuring PHI is much harder: complex access and read/write operations

```
DO IDO=1,NREDD
   INC = INDIR(IWaiting)
   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)
   DLT= REL*( HANB/AM(INC,7) - PHI(INC) )
   PHI(INC) = PHI(INC) + DLT

RESI = RESI + ABS(DLT)
RSUM = RSUM + ABS(PHI(INC))
ENDDO
```

*Fig. 1. Target Loop in RBgauss*
DASSAULT application solves the Navier-Stokes equation using computational fluid dynamics based on an iterative solver.

The hottest subroutine in Dassault application is Eufluxm.

The target loop in Eufluxm is shown in Fig. 2.

Bad access (strides) to arrays.
Impact of load/store instructions on Eufluxm subroutine

Transformed binaries generated by DECAN

Cycles
DASSAULT application - Grouping SSE memory instructions that access to the same base address (ompu & ompl arrays)
When replacing one load at the same time, there is no effect of the replaced load.

When replacing all loads, performance is improved by a factor of 3 → some “dependent” loads seem to be the bottleneck.

Grouping shows that most of the performance loss is due to **access to two 3D arrays**: ompu & ompl

Conclusion: **ompu & ompl access seems to be the bottleneck**!

A memory trace tool is used to detect how these arrays are accessed: Ompl & ompu are accessed with a LARGE STRIDE! (iterating on the wrong dimension)

Only ompu and ompl need to be restructured
Increasing DECAN functionalities (1)

• Compare performance impact with microbenchmark results
  ▪ Use to detect/guess operand location: L1, L2, L3, RAM
  ▪ Use to evaluate prefetch efficiency

• Go beyond nopping:
  ▪ Instead of a NOP use a register move (pay attention to dependencies)
  ▪ Instead of a NOP, perform an access to a given (invariant memory location on the stack (keep cache access latency impact)

• NOP other instructions than memory operations
  ▪ Arithmetic complex instructions: divide, square root
  ▪ Analyze impact of out of order

A Decremental Analysis Tool for a Fine-Grained Bottleneck Detection
Increasing DECAN functionalities (2)

- NOP branches
  - Two variants: force fall through or taken branch
  - Analyze impact of branch misprediction

- Detection of multicore issues:
  - Detection of false sharing
  - Detection of contention
DECAN limitations (1)

- Dealing with side effects:
  - “Nopping” instructions is not exactly neutral
  - Large set of experiments allows to “recoup”

- SEMANTICS is lost
  - From a performance point of view, limited importance but pay attention to some corner cases
  - Some experiments in the DECAN series can crash: for example NOP the access to indirection vectors

- Dealing with If within loop bodies
  - Typical case: if (A(I)) > 0) THEN .... ELSE
  - Nopping A(I) is equivalent to Nopping the branch
  - DECAN provides info but care has to be taken
DECAN limitations (2)

- DECAN is a microscope: applicable to loops only
  - Needs to be coupled with good profiling

- Measurement accuracy
  - Let us think of a loop with 200 vector loads,
  - Some experiments in the DECAN series can crash: for example NOP the access to indirection vectors
DECAN Vs VTune

- VTune is an event-based sampling tool that uses hardware counters.
- VTune collects data from the processor using timer interrupts.
- RBgauss and EUFLUXm routines are profiled with VTune (Fig. 1 & Fig. 2).
- VTune detects a large set of instructions that are not all delinquent.
- This inaccuracy is inherent to any sampling scheme.
- Sampling is useful for a broad diagnostic when DECAN gives a more precise bottleneck detection.
<table>
<thead>
<tr>
<th>Addr</th>
<th>Lin Num</th>
<th>Source</th>
<th>CPU_CLK_UNHALTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2BFE</td>
<td>87</td>
<td>movss -4 (%rcx, %r15, 4), %xmm8</td>
<td>0.20%</td>
</tr>
<tr>
<td>0x2C05</td>
<td>87</td>
<td>mulss (%rdi, %r15, 4), %xmm8</td>
<td>4.87%</td>
</tr>
<tr>
<td>0x2C0B</td>
<td>88</td>
<td>movss -4 (%r13, %r15, 4), %xmm3</td>
<td>0.68%</td>
</tr>
<tr>
<td>0x2C12</td>
<td>88</td>
<td>mulss -8 (%rdi, %r15, 4), %xmm3</td>
<td>5.13%</td>
</tr>
<tr>
<td>0x2C19</td>
<td>89</td>
<td>movss -4 (%r12, %r15, 4), %xmm4</td>
<td>0.70%</td>
</tr>
<tr>
<td>0x2C20</td>
<td>89</td>
<td>mulss -4 (%r11, %r15, 4), %xmm4</td>
<td>4.86%</td>
</tr>
<tr>
<td>0x2C27</td>
<td>90</td>
<td>movss -4 (%r9, %r15, 4), %xmm5</td>
<td>1.21%</td>
</tr>
</tbody>
</table>

**Fig. 1. RBgauss profiled with VTune**
<table>
<thead>
<tr>
<th>Address</th>
<th>Line Num</th>
<th>Source</th>
<th>CPU_CLK_UNHALTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2F06</td>
<td>82</td>
<td>eufluxm_+0x456: movsd -8(%rbx, %rd</td>
<td>1.59%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movsd -8(%rbp, %r15, 1), %xmm0</td>
<td>1.99%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mulsd %xmm2, %xmm0</td>
<td>23.04%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addsd -8(%r12, %rdi, 1), %xmm0</td>
<td>3.92%</td>
</tr>
<tr>
<td>0x2F13</td>
<td>82</td>
<td>eufluxm_+0x456: movsd -8(%rbx, %rd</td>
<td>8.75%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movsd -8(%rbp, %r9, 1), %xmm3</td>
<td>0.03%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movsd %xmm0, -8(%r12, %rdi, 1)</td>
<td>0.82%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mulsd %xmm2, %xmm3</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 2. EUFLUXm profiled with VTune**
Conclusion & Future Work

- DECAN: a tool for automatic decremental performance analysis.
- DECAN identifies delinquent memory operations.
- DECAN gets an estimate of potential performance gain.
- Test DECAN on more applications.
- Improve user feedback: synthesis of DECAN results.
- Extend DECAN to address branch instructions to detect miss-prediction.

A Decremental Analysis Tool for a Fine-Grained Bottleneck Detection
• EXATEC LAB grand opening will take place on October 25th at UVSQ in Versailles

• You are all invited and welcome!!

• See http://www.uvsq.fr : front page
Questions ?
Testing on Intel Core i7

\[ a[i] = b[i - \text{offset}] \]; \text{sizeof}(a,b) = 512\text{Ko}

core i7
All optimization on Intel Core 2 Duo

\[ a[i] = b[i - \text{offset}] \] ; \text{sizeof}(a,b) = 512\text{Ko}

Core 2 Duo
Introduction

- **Optimization process:**
  - Gathering data (ie. code characterization)
  - Diagnosing the problem
  - Prescribing a solution

- **Tedious process**
  - Complex modern processors
  - Limited existing methodologies
  - Performance counters not up to the job

- **Characterization process**
  - Code analysis to extract code characteristics
  - Applying different types of code analysis
  - Get different views of the code behavior