DECAN: Decremental Performance Analysis Tool via Binary Patching

EXATEC-LAB
CEA GENCI INTEL UVSQ

UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

Souad Koliaï
Ph.D. Candidate
University of Versailles
souad.koliai@prism.uvsq.fr

Grigori Fursin
Doctor
University of Versailles
grigori.fursin@prism.uvsq.fr

Tipp Moseley
Doctor
Google Corp.
tipp@google.com

William Jalby
Professor
University of Versailles
william.jalby@prism.uvsq.fr

Optimization process

- Gathering data (ie. code characterization);
- Diagnosing the problem;
- Prescribing a solution.

Tedious process

- Complex modern processors;
- Limited existing methodologies;
- Performance counters not understandable.

Characterization process

- Code analysis to extract code characteristics;
- Applying different types of the code analysis;
- Get different views of the code behavior.

What is DECAN?

- DECAN is a tool for a fine-grained detection of bottlenecks (ie. assembly instruction level);
- DECAN focuses on the hot region of the application;
- DECAN performs on a binary level;
- DECAN uses Pin to patch the binary.

DECAN general concept

1. Measure the original binary;
2. Patch (ie.remove) the memory access instructions in the original binary;
3. New binary is generated for each patch
4. Measure new binaries;
5. Measure are represented in a CSV file.

DECAN: Instruction Removal

- DECAN focuses on SSE memory instructions (loads/stores)
- DECAN replaces the memory instruction by a nop with variable size:

```
movaps (%rsi), %xmm1 -> nop r/m
movaps  %xmm1, (%rsi)-> nop r/m
```

- Each patched instruction generates a new binary
- DECAN generates new binaries in 6 different ways: one load, one store, all loads, all stores, all loads/stores, grouping.

How DECAN removes the SSE memory instructions?

Original code

```
Loop:
  movsd (%rsi,%rax),%xmm1
  mulsd  %xmm0,%xmm1
  addsd  (%rsi,%rax),%xmm1
  movsd %xmm1,(%rsi,%rax)
  inc       %rax
  jb         Loop
```

One load

- Replace each SSE load with a nop
- Each replacement generates a new binary

One store

- Replace each SSE store with a nop
- Each replacement generates a new binary

All loads

- Replace all SSE loads with nops and generate a new binary

All stores

- Replace all SSE stores with nops and generate a new binary

All loads/stores

- Replace all SSE loads and stores with nops and generate a new binary

Grouping

- Remove (ie.replace by nop) all loads that access to the same base address

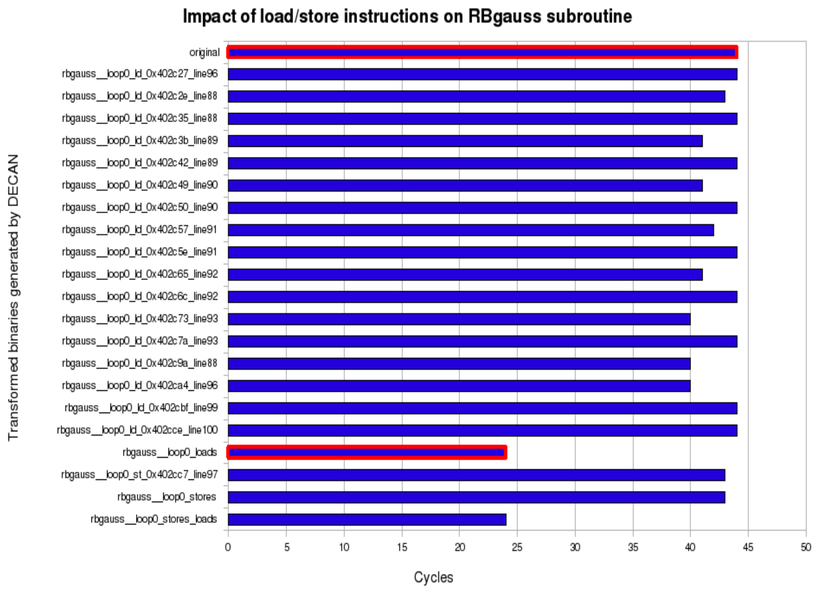
DECAN application on real-life applications

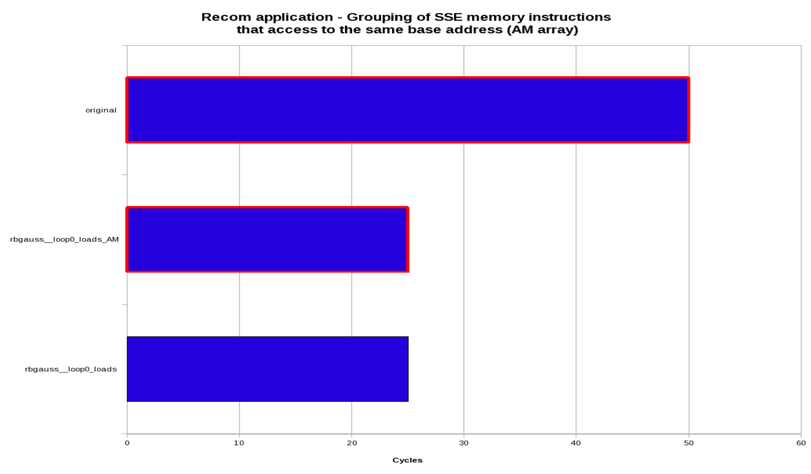
RBgauss subroutine

```
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
```

Matvec subroutine

```
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)
    enddo
  enddo
enddo
```





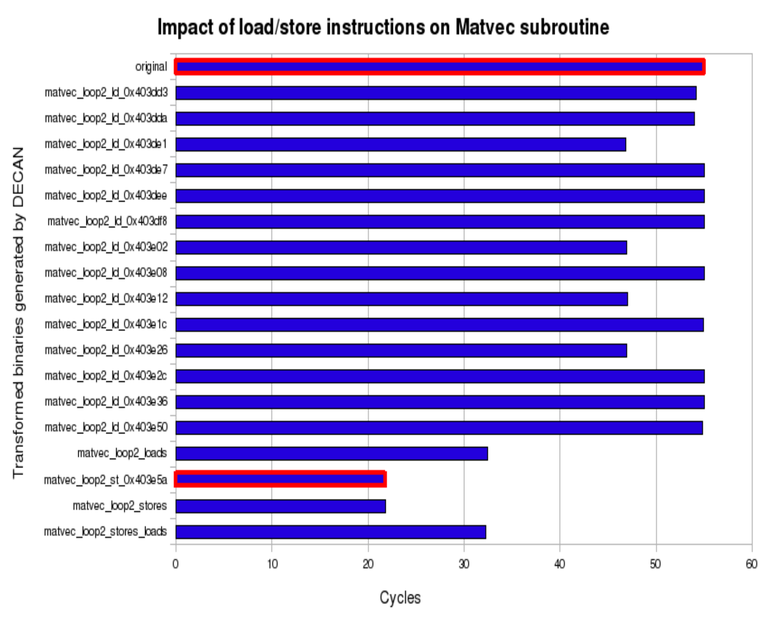
RBgauss subroutine

- Removing all loads improves performance.
- Removing a set of loads accessing to the same base address (AM array) --> lower bound achieved.
- The bottleneck is the AM access and only AM.
- Stride 2 access that causes bottleneck only in AM access.

--> Bottleneck precisely pinpointed, applying an optimization on AM access.

Matvec subroutine

- Removing one store improves the performance.
- The store is the bottleneck.
- The store corresponds to the store of `vhilf`.
- There is a conflict between the store of `vhilf` and some loads (`acx & temp`).
- It is a problem of 4K-aliasing: `vhilf`, `acx`, and `temp` have the same address modulo 4K.
- DECAN detects precisely that the bottleneck is the store instruction.



Futur work

- Test DECAN on more applications (SPEC 2006).
- Improve user's feedback: synthesis of DECAN's results.
- Extend DECAN to address branch instructions to detect miss-prediction.