PENE: Pin Enabled Numerical Exploration

Prepared by Malak Elflitty
Supervisors:
Wilfried Kirschenmann
Florian Lemaitre
Jérôme Gurhem

Plan

Outline

Round-off errors source of problems in computation

- non representable numbers
  →round-off errors
  →accumulation of many round-off errors
- subtraction of close numbers
  →cancellation
- subtraction of distant numbers
  →absorption

Interflop Project

Presentation

- is carried by 8 teams
- aims to provide a common platform to analyze and control the cost of floating-point behavior on programs

Interflop members developed tools that analyze floating-point behavior : Cadna [2], Verificarlo [1] and Verrou [3]

Figure – Interflop consortium

Analysis by changing arithmetic

- use an arithmetic for calculation other than floating-point arithmetic to analyze errors. For example : Stochastic Arithmetic

Stochastic Arithmetic

- Repeat each arithmetic floating-point operation N times with random rounding mode
- Model the uncertainties on the results of floating-point operations as random variables

Outline

Need to change arithmetic

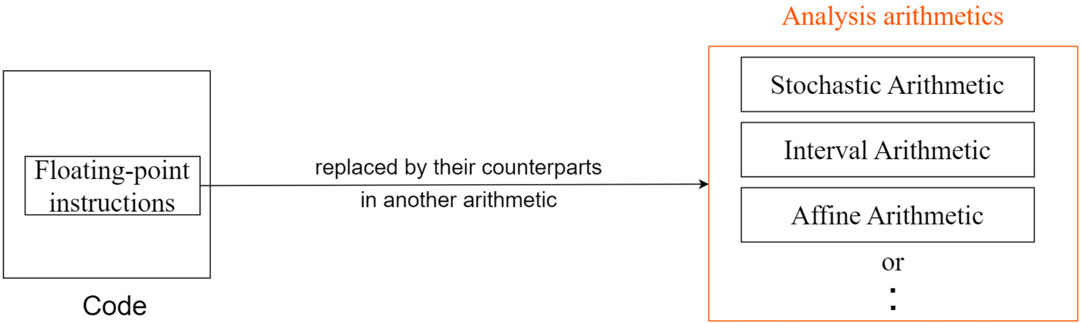


Figure – Replacing floating-point arithmetic in a code

Front-end and Back-end

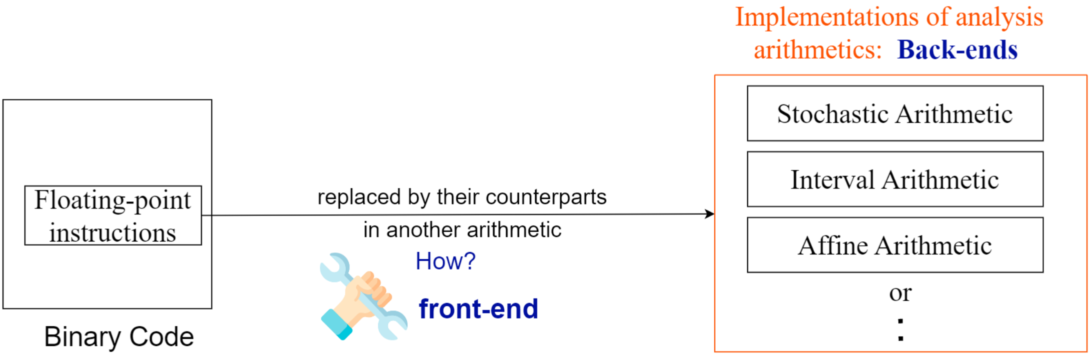


Figure – Front-end and back-end

Role of PENE

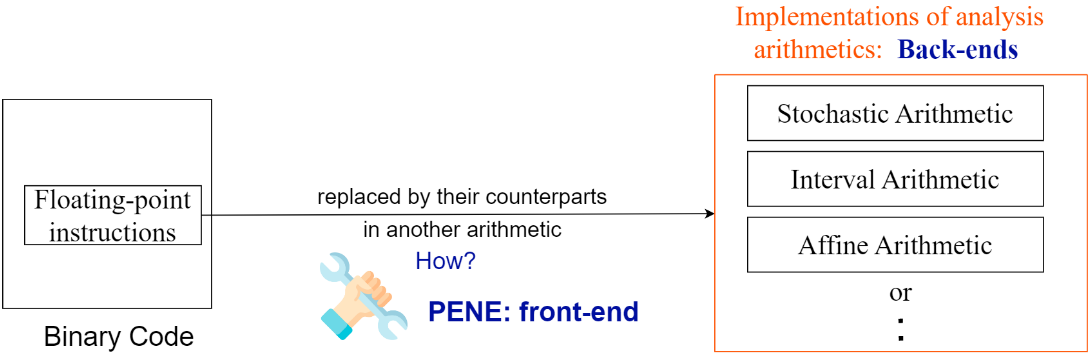


Figure – PENE : front-end dealing with executables changing arithmetic

PENE : a tool to instrument floating-point instructions

- modification of executable code
  $\Longrightarrow$ No need to recompile source code
- supports Windows and Linux

Analysis and instrumentation codes

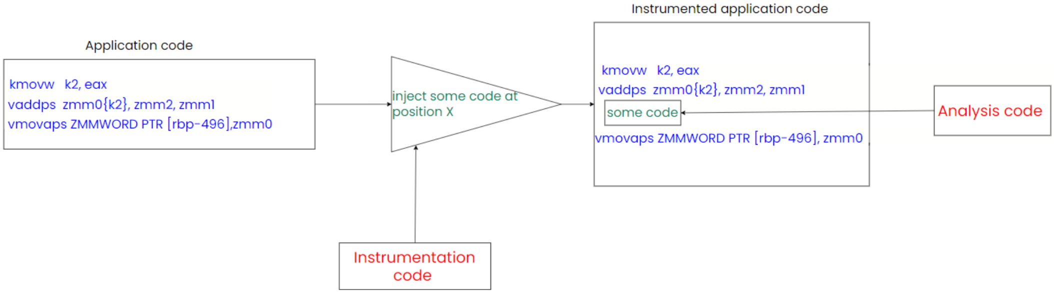


Figure – Instrumentation of a code

Connected back-ends

- IEEE back-end
- Verrou back-end developed by François Févotte and Bruno Lathuilière.
  Changing rounding mode :
  - deterministic rounding
  - stochastic rounding

Outline

Role of instrumentation code

- intercept floating-point instructions in a code
- call back-end functions to replace them

Number of variants of instructions to handle

- need to handle floating-point instructions with all their variants

Number of variants

921 variants of floating-point instructions

VADDPS xmm1, xmm2, xmm3
VADDPS xmm1,xmm2, mem128
VADDPS ymm1,ymm2, ymm3
VADDPS ymm1,ymm2, mem256

Table – Variants of instruction VADDPS

Instrumentation code can not be manually written

Code – Example of instrumentation code for only two instruction variants

```
case xed_iform_enum_t::XED_IFORM_ADDSS_XMMss_MEMss:{
INS_InsertCall(ins,IPOINT_BEFORE,(AFUNPTR)call_backend_fct<float, OPERATION_IMPL::add_float>,
IARG_REG_CONST_REFERENCE, INS_OperandReg(ins,0),
IARG_MEMORYREAD_EA,
IARG_REG_REFERENCE, INS_OperandReg(ins,0),
IARG_PTR, backend_ctx,
IARG_UINT32,1,
IARG_END);
INS_Delete(ins);
break;
}
case xed_iform_enum_t::XED_IFORM_VADDSD_XMMdq_XMMdq_XMMq:{
INS_InsertCall(ins,IPOINT_BEFORE,(AFUNPTR)call_backend_fct<double, OPERATION_IMPL::add_double>,
IARG_REG_CONST_REFERENCE, INS_OperandReg(ins,1),
IARG_REG_CONST_REFERENCE, INS_OperandReg(ins,2),
IARG_REG_REFERENCE, INS_OperandReg(ins,0),
IARG_PTR, backend_ctx,
IARG_UINT32,1,
IARG_END);
INS_Delete(ins);
break;
}
```

Implementation of a code generator

instrumentation code generated with python and Jinja

- time and effort saving
- more maintainable code
- more scalable code

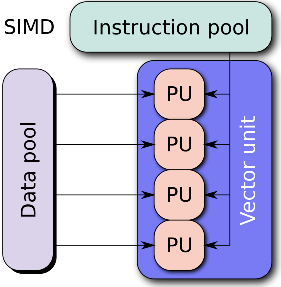
Adding new instructions

- 600 variants of instructions handled by PENE :
  - elementary operations instructions (+,-,x,/) with their variants for SSE2-SSE4.2, AVX and AVX512
  - FMA instructions *Fused-Multiply Add* with all their variants

Outline

Approaches to instrument SIMD instructions

- *Single Instruction Multiple Data* : simultaneous processing of multiple data with a single instruction.
- No vectorized back-end $\Longrightarrow$ Need to devectorize SIMD instructions
  - iterate over vectors elements
  - call back-end function on each pair of elements la fonction



Source : *Hardware times*

Figure – How SIMD instructions operate

Approaches to instrument SIMD instructions

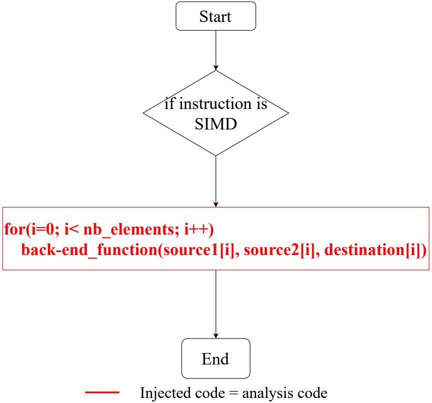


Figure – Devectorization inside analysis code

Approaches to instrument SIMD instructions

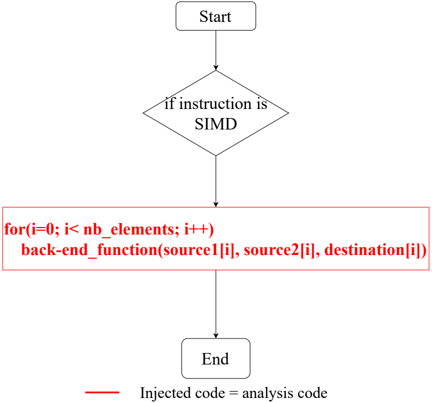


Figure – Devectorization inside analysis code

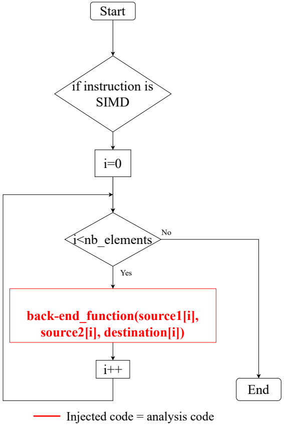


Figure – Devectorization outside analysis code

Comparison of approaches

- performance advantages : helping Pin inline analysis code
- small routines generally inlined
- comparing the latency by measuring execution times

Conditions of measurements of execution times

- execution time of a calculation code : matrix inversion with Gauss-Jordan
- inverted matrix : 200x200
- measured time : execution time of 1000 inversions.

$$[\,A \quad I\,] \equiv \begin{bmatrix} a_{11} & \cdots & a_{1\,n} & 1 & 0 & \cdots & 0 \\ a_{21} & \cdots & a_{2\,n} & 0 & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n\,1} & \cdots & a_{n\,n} & 0 & 0 & \cdots & 1 \end{bmatrix}$$

Source : *Wolfram Mathworld mathematics resource*

Figure – Inversion of a matrix using Gauss-Jordan

Conditions of measurements of execution times

Code compiled with :

- -O3 + -msse4
- -O3 + -mavx

Means of execution :

- vanilla
- PENE without instrumentation
- PENE : instrumentation with back-end IEEE
- PENE : instrumentation with back-end Verrou, nearest rounding mode
- PENE : instrumentation with back-end Verrou, random rounding mode

Comparison of approaches

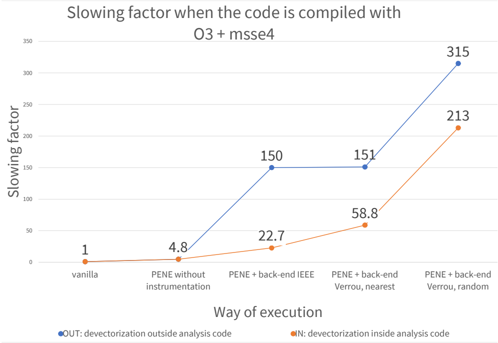


Figure – Comparison of the approaches for the case *O3* + *msse4*

Comparison of approaches

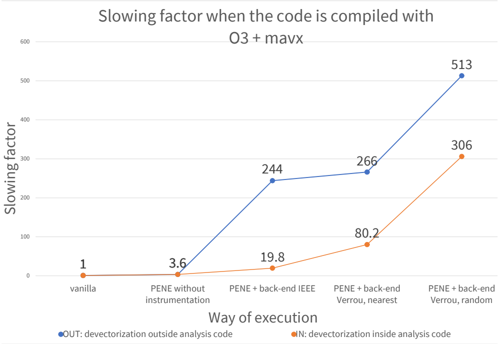


Figure – Comparison of approaches for the case *O3* + *mavx*

Outline

Unitary tests

- counting
- by replacing with two debug back-ends
  - the first one replaces additions with multiplications.
  - the second one alters the last 4 bits of the results of each operation by a specific pattern.

Test back-end

```
struct interflop_backend_interface_t {

void add_float(float, float, float*, void*);                        //pattern 0x1 = 0001
void sub_float(float, float, float*, void*);                        //pattern 0x2 = 0010
void mul_float(float, float, float*, void*);                        //pattern 0x3 = 0011
void div_float(float, float, float*, void*);                        //pattern 0x4 = 0100
void madd_float(float , float , float , float *, void *);           //pattern 0X5 = 0101
void add_double(double, double, double*, void*);                    //pattern 0x6 = 0110
void sub_double(double, double, double*, void*);                    //pattern 0x7 = 0111
void mul_double(double, double, double*, void*);                    //pattern 0x8 = 1000
void div_double(double, double, double*, void*);                    //pattern 0x9 = 1001
void madd_double(double , double , double , double *, void *);      //pattern 0xa = 1010

};
```

Figure – Patterns of back-end test functions

Outline

Conditions of measuring of execution times

Code compiled with :

- -msse4
- -mavx
- -mfma
- -mavx512f

each combined with

- -O2
- -O3

Means of execution :

- vanilla
- PENE without instrumentation
- PENE : instrumentation with back-end IEEE
- PENE : instrumentation with back-end Verrou, nearest rounding mode
- PENE : instrumentation with back-end Verrou, random rounding mode

Number of instrumented instructions

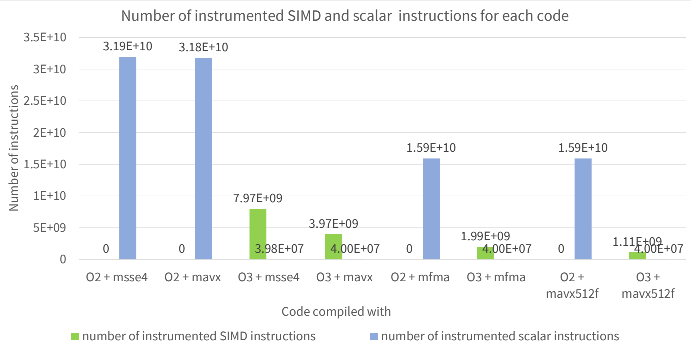


Figure – Number of instrumented instructions SIMD and scalar for each code

Evaluation of instrumentation overhead

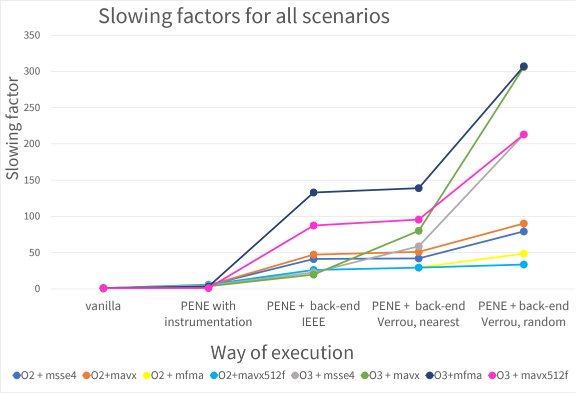


Figure – Slowing factors for all codes for while executed through different ways

Instrumentation overhead PENE versus Verrou

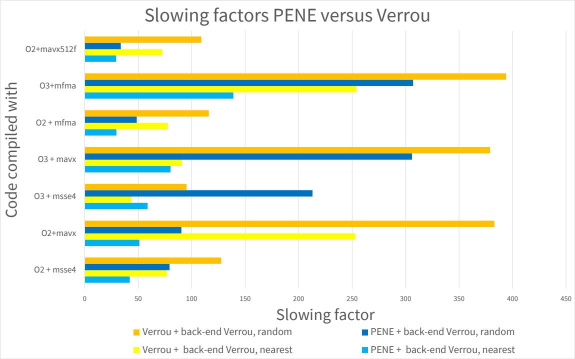


Figure – Comparison of slowing factors for PENE and Verrou

Outline

Exclusion filters

Exclusion

- based on function symbol
- contains all the symbols to be included with the library path

Next step : Implement delta-debugging algorithm

Perspectives

- have different versions of back-ends compiled differently

```
Analyze Analysis function from /XXXX/XXXXXX/XXXXX/XXXXXXX/XXXXXXX/XXXXXXXX/sse.h:287 for inlining, function has 4 instructions:
[tid:25746]    572 0x000000000 0x00007f396de619f0 nop edx, edi
[tid:25746]    451 0x000000000 0x00007f396de619f4 vmovss xmm0, dword ptr [rdi]
[tid:25746]    506 0x000000000 0x00007f396de619f8 vdivss xmm0, xmm0, dword ptr [rsi]
[tid:25746]    450 0x000000000 0x00007f396de619fc vmovss dword ptr [rdx], xmm0
[tid:25746]    NOT INLINED in this instance in order to avoid possible avx-sse transition penalty
    [tid:25746]    NOT INLINED
[tid:25746]    INSERT_BEFORE_INSTRUCTION    424 0x000056063b88566a divss xmm4, dword ptr [rcx]
```

Figure – Inling report

Thank you !

References

[1] Christophe DENIS, Pablo De Oliveira CASTRO et Eric PETIT. « Verificarlo : Checking floating point accuracy through monte carlo arithmetic ». In : *arXiv preprint arXiv :1509.01347* (2015).

[2] Pacôme EBERHART et al. « High Performance Numerical Validation using Stochastic Arithmetic ». In : 21 (déc. 2015).

[3] François FÉVOTTE et Bruno LATHUILIÈRE. « Verrou : Assessing floating-point accuracy without recompiling ». In : (2016).