*SMAI 2011, Guidel (France), May 23–27 2011*

# Validated performance of accurate algorithms

**Bernard Goossens, Philippe Langlois, David Parello**

DALI Research Project, University of Perpignan Via Domitia
LIRMM Laboratory, CNRS – University Montpellier 2, France.

**DALI**, Digits, Architectures
et Logiciels Informatiques

LIRMM

cnrs

UPVD
Université de Perpignan Via Domitia

# Context and motivation

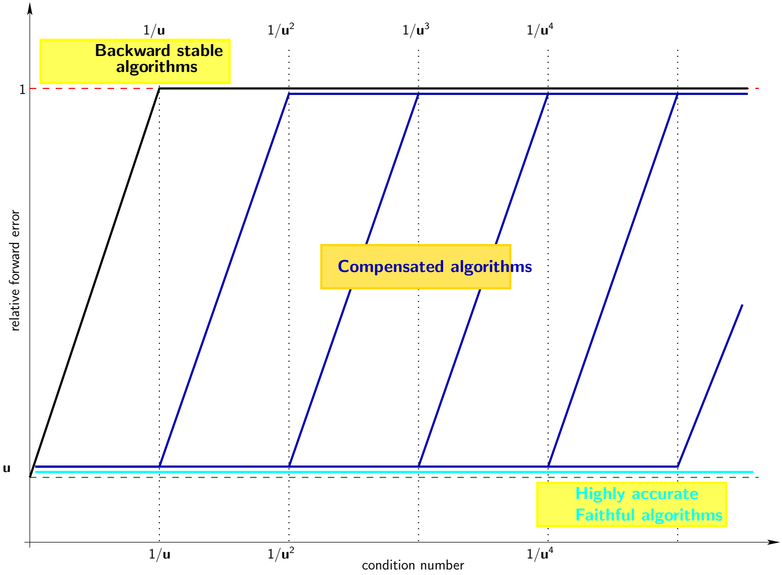
Context: Floating point computation using IEEE-754 arithmetic (64 bits)

Aim: Improve and validate the accuracy of numerical algorithms . . .
. . . without sacrificing the running-time performances

Improving accuracy:
Why ? result accuracy ≈ condition number × machine precision
How ? more bits

- double-double (128) or quad-double librairies (256)
- MPFR (arbitrary # bits, fast for 256+)
- Compensated algorithms

# Computed accuracy is constraint by the condition number



# Compensated algorithms: accurate and fast

## Compensated algorithms

- summation and dot product: Knuth (65), Kahan (66), . . . , Ogita-Rump-Oishi (05,08)
- polynomial evaluation: Horner (Langlois-Louvet, 07), Clenshaw, De Casteljau (Hao et al., 11)
- triangular linear systems: (Langlois-Louvet, 08)

These algorithms are fast in terms of measured computing time

- Faster than other existing solutions: double-double, quad-double, MPFR
  Question: how to trust such claim?
- Faster than the theoretical complexity that counts floating-point operations
  Question: how to explain and verify such claim —at least illustrate?

# Flop counts and running-times are not proportional

A classic problem: I want to double the accuracy of a computed result while running as fast as possible?

A classic answer:

| Metric | Eval | AccEval1 | AccEval2 |
|---|---|---|---|
| Flop count | 2n | $22n + 5$ | $28n + 4$ |
| Flop count ratio | 1 | $\approx 11$ | $\approx 14$ |
| Measured #cycles ratio | 1 | 2.8 – 3.2 | 8.7 – 9.7 |

Flop counts and running-times are not proportional. Why? Which one trust?

# Running-time measures: details

Average ratios for polynomials of degree 5 to 200

Working precision: IEEE-754 double precision

| | | $\frac{\text{CompHorner}}{\text{Horner}}$ | $\frac{\text{DDHorner}}{\text{Horner}}$ | $\frac{\text{DDHorner}}{\text{CompHorner}}$ |
|---|---|---|---|---|
| Pentium 4, 3.00 GHz | GCC 4.1.2 | 2.8 | 8.5 | 3.0 |
| (x87 fp unit) | ICC 9.1 | 2.7 | 9.0 | 3.4 |
| (sse2 fp unit) | GCC 4.1.2 | 3.0 | 8.9 | 3.0 |
| (sse2 fp unit) | ICC 9.1 | 3.2 | 9.7 | 3.4 |
| Athlon 64, 2.00 GHz | GCC 4.1.2 | 3.2 | 8.7 | 3.0 |
| Itanium 2, 1.4 GHz | GCC 4.1.1 | 2.9 | 7.0 | 2.4 |
| | ICC 9.1 | 1.5 | 5.9 | 3.9 |

Results vary with a factor of 2

Life-period for the significance of these computing environments?

# How to trust non-reproducible experiment results?

## Measures are mostly non-reproducible

- The execution time of a binary program varies, even using the same data input and the same execution environment.

## Why? Experimental uncertainties

- spoiling events: background tasks, concurrent jobs, OS interrupts
- non deterministic issues: instruction scheduler, branch predictor
- external conditions: temperature of the room (!)
- timing accuracy: no constant cycle period on modern processors (i7, . . . )

## Uncertainty increases as computer system complexity does

- architecture issues: multicore, many/multicore, hybrid architectures
- compiler options and its effects

# How to read the current literature?

Lack of proof, or at least of reproducibility

> *Measuring the computing time of summation algorithms in a high-level language on today's architectures is more of a hazard than scientific research.* S.M. Rump (SISC, 2009)

The picture is blurred: the computing chain is wobbling around

> *If we combine all the published speedups (accelerations) on the well known public benchmarks since four decades, why don't we observe execution times approaching to zero?* S. Touati (2009)

# Outline

1. Accurate algorithms : why ? how ? which ones ?

2. How to choose the fastest algorithm?

3. The PerPI Tool
    - Goals and principles
    - What is ILP?

4. The PerPI Tool: outputs and first examples

5. Conclusion

# Highlight the potential of performance

## General goals

- Understand the algorithm and architecture interaction
- Explain the set of measured running-times of its implementations
- Abstraction *w.r.t.* the computing system for performance prediction and optimization
- Reproducible results in time and in location
- Automatic analysis

## Our context

- Objects: accurate and core-level algorithms: XBLAS, polynomial evaluation
- Tasks: compare algorithms, improve the algorithm while designing it, chose algorithms → architecture, optimize algorithm → architecture

# The PerPI Tool: principles

## Abstract metric: Instruction Level Parallelism

- ILP: the potential of the instructions of a program that can be executed simultaneously
- #IPC for the Hennessy-Patterson ideal machine
- Compilers and processors exploits ILP: superscalar out-of-order execution
- Thin grain parallelism suitable for single node analysis

# What is ILP?

A synthetic sample: $e = (a+b) + (c+d)$

x86 binary

```
    ...
i1  mov     eax,DWP[ebp-16]
i2  mov     edx,DWP[ebp-20]
i3  add     edx,eax
i4  mov     ebx,DWP[ebp-8]
i5  add     ebx,DWP[ebp-12]
i6  add     edx,ebx
    ...
```

# What is ILP?

A synthetic sample: $e = (a+b) + (c+d)$

x86 binary

```
    ...
i1  mov     eax,DWP[ebp-16]
i2  mov     edx,DWP[ebp-20]
i3  add     edx,eax
i4  mov     ebx,DWP[ebp-8]
i5  add     ebx,DWP[ebp-12]
i6  add     edx,ebx
    ...
```

Instruction and cycle counting

# What is ILP?

A synthetic sample: e = (a+b) + (c+d)

x86 binary

```
   ...
i1 mov     eax,DWP[ebp-16]
i2 mov     edx,DWP[ebp-20]
i3 add     edx,eax
i4 mov     ebx,DWP[ebp-8]
i5 add     ebx,DWP[ebp-12]
i6 add     edx,ebx
   ...
```

Instruction and cycle counting

Cycle 0: i1 i2 i4

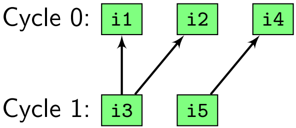
# What is ILP?

A synthetic sample: $e = (a+b) + (c+d)$

x86 binary

```
    ...
i1  mov   eax,DWP[ebp-16]
i2  mov   edx,DWP[ebp-20]
i3  add   edx,eax
i4  mov   ebx,DWP[ebp-8]
i5  add   ebx,DWP[ebp-12]
i6  add   edx,ebx
    ...
```

Instruction and cycle counting



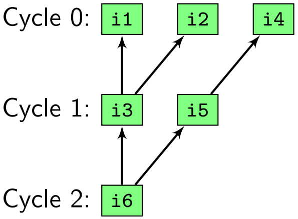
# What is ILP?

A synthetic sample: $e = (a+b) + (c+d)$

x86 binary

```
      ...
i1    mov    eax,DWP[ebp-16]
i2    mov    edx,DWP[ebp-20]
i3    add    edx,eax
i4    mov    ebx,DWP[ebp-8]
i5    add    ebx,DWP[ebp-12]
i6    add    edx,ebx
      ...
```

Instruction and cycle counting



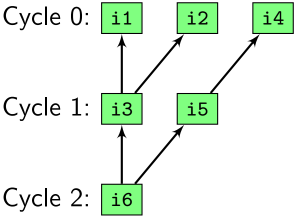
# What is ILP?

A synthetic sample: $e = (a+b) + (c+d)$

x86 binary

```
    ...
i1  mov     eax,DWP[ebp-16]
i2  mov     edx,DWP[ebp-20]
i3  add     edx,eax
i4  mov     ebx,DWP[ebp-8]
i5  add     ebx,DWP[ebp-12]
i6  add     edx,ebx
    ...
```

Instruction and cycle counting

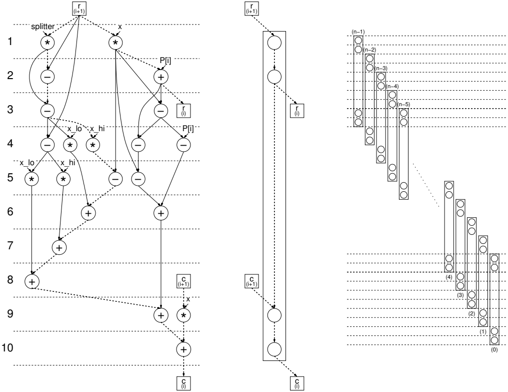


$$\# \text{ of instructions} = 6,\ \# \text{ of cycles} = 3$$

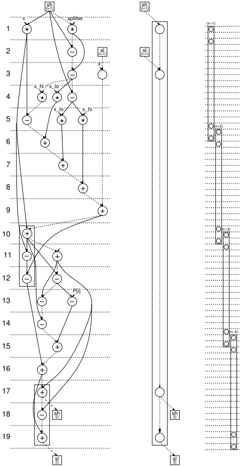
$$\text{ILP} = \# \text{ of instructions} / \# \text{ of cycles} = 2$$

# ILP explains why compensated algorithms are fast

| | AccEval | AccEval2 |
|---|---|---|
| ILP: | $\approx 11$ | 1.65 |



(a) (b) (c)



(a) (b) (c)

# The PerPI Tool: principles

## From ILP analysis to the PerPI tool

- 2007: successful previous pencil-and-paper ILP analysis [PhL-Louvet,2007]
- 2008: prototype within a processor simulation platform (PPC asm)
- 2009: PerPI to analyse and visualise the ILP of x86-coded algorithms

## PerPI

- Pintool (http://www.pintool.org)
- Input: x86 binary file
- Outputs: ILP measure, IPC histogram, data-dependency graph

# Outline

1. Accurate algorithms : why ? how ? which ones ?
2. How to choose the fastest algorithm?
3. The PerPI Tool
4. The PerPI Tool: outputs and first examples
5. Conclusion

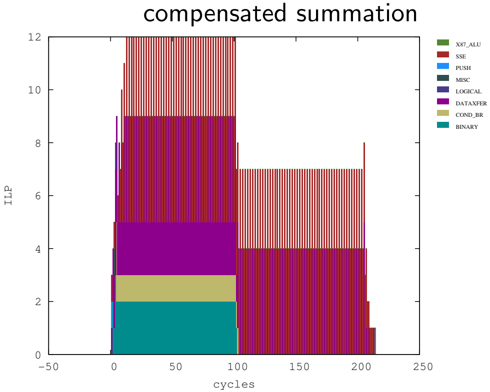
# Simulation produces reproducible results

```
     start : _start
          start : .plt
               start : __libc_csu_init
                    start : _init
                         start : call_gmon_start
                         stop : call_gmon_start::I[13]::C[9]::ILP[1.44444]
                         start : frame_dummy
                         stop : frame_dummy::I[7]::C[3]::ILP[2.33333]
                         start : __do_global_ctors_aux
                         stop : __do_global_ctors_aux::I[11]::C[6]::ILP[1.83333]
                    stop : _init::I[41]::C[26]::ILP[1.57692]
               stop : __libc_csu_init::I[63]::C[39]::ILP[1.61538]
               start : main
                    start : .plt
                         start : .plt
                              start : Horner
                              stop : Horner::I[5015]::C[2005]::ILP[2.50125]
                              start : Horner
                              stop : Horner::I[5015]::C[2005]::ILP[2.50125]
                              start : Horner
                              stop : Horner::I[5015]::C[2005]::ILP[2.50125]
                         stop : main::I[20129]::C[7012]::ILP[2.87065]
                         start : _fini
                              start : __do_global_dtors_aux
                              stop : __do_global_dtors_aux::I[11]::C[4]::ILP[2.75]
                         stop : _fini::I[23]::C[13]::ILP[1.76923]

Global ILP ::I[20236]::C[7065]::ILP[2.86426]
```

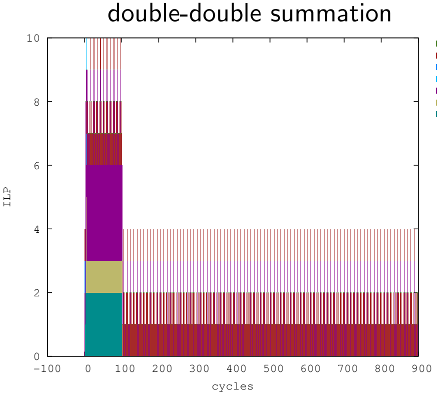
# Profile results to compare two algorithms

```
start :          _start   (depth: 1 rtn_s_d: 0)
          start : __libc_csu_init   (depth: 2 rtn_s_d: 0)
               start :            _init   (depth: 3 rtn_s_d: 0)
                    start : call_gmon_start   (depth: 4 rtn_s_d: 0)
                    stop  : call_gmon_start   (depth: 4 rtn_s_d: 0)   I[13]::C[9]::ILP[1.44444]
                    start :     frame_dummy   (depth: 4 rtn_s_d: 0)
                    stop  :     frame_dummy   (depth: 4 rtn_s_d: 0)   I[7]::C[3]::ILP[2.33333]
                    start : __do_global_ctors_aux   (depth: 4 rtn_s_d: 0)
                    stop  : __do_global_ctors_aux   (depth: 4 rtn_s_d: 0)   I[11]::C[6]::ILP[1.8
               stop  :            _init   (depth: 3 rtn_s_d: 0)   I[41]::C[26]::ILP[1.57692]
          stop  : __libc_csu_init   (depth: 2 rtn_s_d: 0)   I[63]::C[39]::ILP[1.61538]
          start :            main   (depth: 2 rtn_s_d: 0)
               start :          Horner   (depth: 3 rtn_s_d: 0)
               stop  :          Horner   (depth: 3 rtn_s_d: 0)   I[519]::C[206]::ILP[2.51942]
               start :      CompHorner   (depth: 3 rtn_s_d: 0)
               stop  :      CompHorner   (depth: 3 rtn_s_d: 0)   I[3732]::C[318]::ILP[11.7358]
               start :        DDHorner   (depth: 3 rtn_s_d: 0)
               stop  :        DDHorner   (depth: 3 rtn_s_d: 0)   I[4229]::C[2106]::ILP[2.00807]
          stop  :            main   (depth: 2 rtn_s_d: 0)   I[9062]::C[2509]::ILP[3.6118]
          start :           _fini   (depth: 2 rtn_s_d: 0)
               start : __do_global_dtors_aux   (depth: 3 rtn_s_d: 0)
               stop  : __do_global_dtors_aux   (depth: 3 rtn_s_d: 0)   I[11]::C[4]::ILP[2.75]
          stop  :           _fini   (depth: 2 rtn_s_d: 0)   I[23]::C[13]::ILP[1.76923]

Global ILP    I[9169]::C[2562]::ILP[3.57884]
```

# Histograms to compare two algorithms

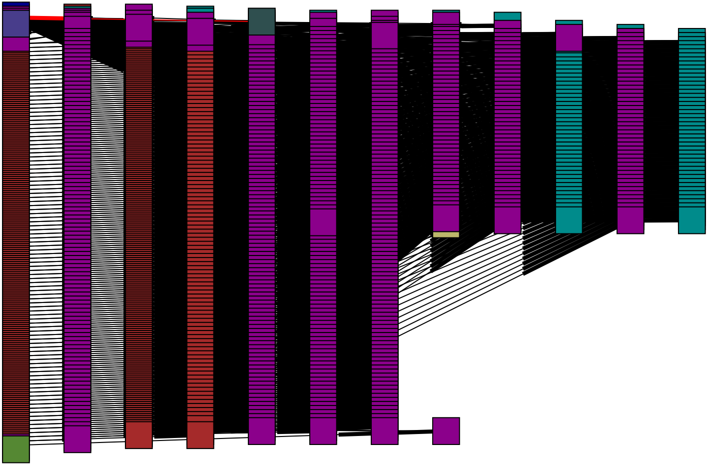


compensated summation



double-double summation

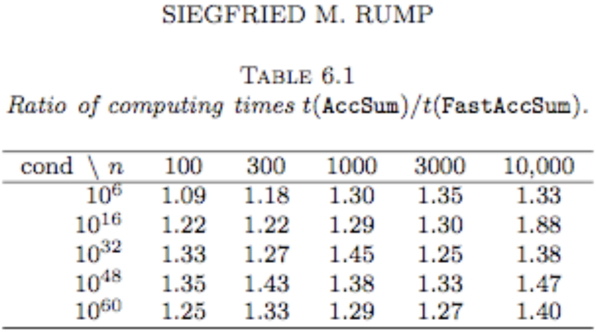
# Visualisation of the instruction dependence graph



# Instruction dependence analysis to compare two algorithms

*Ultimatly Fast Accurate Summation*. S.M. Rump. [SISC,2009]

- New FastAccSum is announced to be faster than AccSum:
- $3n$ vs. $4n$ flop ($\times m$ outer iterations) [SISC,2009]



SIEGFRIED M. RUMP

TABLE 6.1

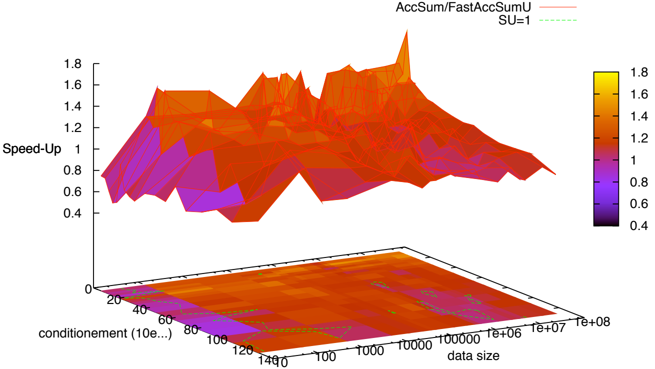
*Ratio of computing times* $t$(`AccSum`)/$t$(`FastAccSum`).

| cond \ $n$ | 100 | 300 | 1000 | 3000 | 10,000 |
|---|---|---|---|---|---|
| $10^{6}$ | 1.09 | 1.18 | 1.30 | 1.35 | 1.33 |
| $10^{16}$ | 1.22 | 1.22 | 1.29 | 1.30 | 1.88 |
| $10^{32}$ | 1.33 | 1.27 | 1.45 | 1.25 | 1.38 |
| $10^{48}$ | 1.35 | 1.43 | 1.38 | 1.33 | 1.47 |
| $10^{60}$ | 1.25 | 1.33 | 1.29 | 1.27 | 1.40 |

# Instruction dependence analysis to compare two algorithms

*Ultimatly Fast Accurate Summation*. S.M. Rump. [SISC,2009]

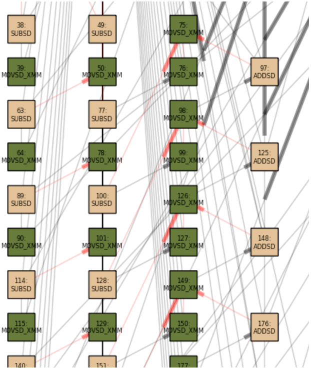
- New FastAccSum is announced to be faster than AccSum:
- $3n$ vs. $4n$ flop ($\times m$ outer iterations) [SISC,2009]

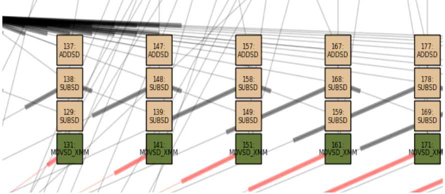


# Instruction dependence analysis to compare two algorithms

*Ultimatly Fast Accurate Summation*. S.M. Rump. [SISC,2009]

- New FastAccSum is announced to be faster than AccSum:
- $3n$ vs. $4n$ flop ($\times m$ outer iterations) [SISC,2009]
- but AccSum benefits for more ILP: PerPI ouputs

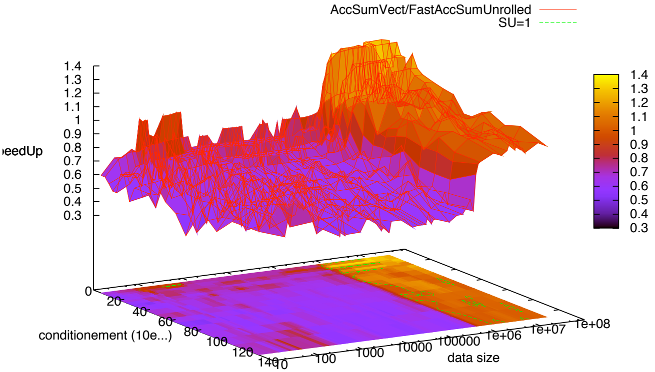




# Instruction dependence analysis to compare two algorithms

*Ultimatly Fast Accurate Summation*. S.M. Rump. [SISC,2009]

- New FastAccSum is announced to be faster than AccSum:
- $3n$ vs. $4n$ flop ($\times m$ outer iterations) [SISC,2009]
- but AccSum benefits for more ILP: PerPI ouputs
- Let's exploit it!

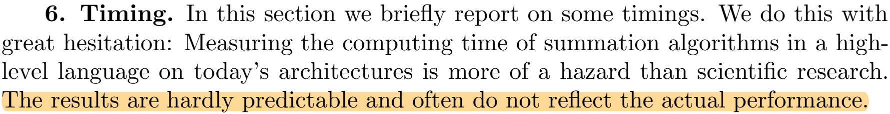


# Instruction dependence analysis to compare two algorithms

*Ultimatly Fast Accurate Summation*. S.M. Rump. [SISC,2009]

- New FastAccSum is announced to be faster than AccSum:

- S.M. Rump is right!



**6. Timing.** In this section we briefly report on some timings. We do this with great hesitation: Measuring the computing time of summation algorithms in a high-level language on today's architectures is more of a hazard than scientific research. The results are hardly predictable and often do not reflect the actual performance.

# This is the end

1. Accurate algorithms : why ? how ? which ones ?
2. How to choose the fastest algorithm?
3. The PerPI Tool
4. The PerPI Tool: outputs and first examples
5. Conclusion

# Conclusions

## PerPI: a software platform to analyze and visualise ILP

- Useful: a detailed picture of the intrinsic behavior of the algorithm
- Reliable: reproducibility both in time and location
- Realistic: correlation with measured ones
- Exploratory tool: gives us the taste of the behavior of our algorithms within "tomorrow" processors
- Optimisation tool: analyse the effect of some hardware constraints

## Cons . . . at the current state

- Work in progress
- Not abstract enough: instruction set dependence (RISC vs. CISC, 3-operand instructions, . . .
- Assembler program or high level programming language?
  IPC vs. FloPC ?

# Current working list

- Improving the post-processing visualisation
- Make PerPI available on-line and usable as black-box