*Numerical Software: Design, Analysis and Verification*
IFIP WG2.5, Santander, July 4–6 2012

# Computing Time of Summation Algorithms:

# Less Hazard and More Scientific Research

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

University of Perpignan Via Domitia, DALI,
University Montpellier 2, LIRMM,
CNRS UMR 5506, France

1 Why measure summation algorithm performance?

2 How to measure summation algorithm performance?

3 ILP and the PerPI Tool

4 Experiments with recent summation algorithms

5 Conclusion

# How to manage accuracy and speed?

## A new "better" algorithm every year since 1999

1965 Møller, Ross

1969 Babuska, Knuth

1970 Nickel

1971 Dekker, Malcolm

1972 Kahan, Pichat

1974 Neumaier

1975 Kulisch/Bohlender

1977 Bohlender, Mosteller/Tukey

1981 Linnaimaa

1982 Leuprecht/Oberaigner

1983 Jankowski/Semoktunowicz/-Wozniakowski

1985 Jankowski/Wozniakowski

1987 Kahan

1991 Priest

1992 Clarkson, Priest

1993 Higham

1997 Shewchuk

1999 Anderson

2001 Hlavacs/Uberhuber

2002 Li et al. (XBLAS)

2003 Demmel/Hida, Nievergelt, Zielke/Drygalla

2005 Ogita/Rump/Oishi, Zhu/Yong/Zeng

2006 Zhu/Hayes

2008 Rump/Ogita/Oishi

2009 Rump, Zhu/Hayes

2010 Zhu/Hayes

# Accuracy of the floating point summation

## Precision

- $\mathbf{u}$= arithmetic precision
- $\mathbf{u} = 2^{-53} \approx 10^{-16}$ for b64 in IEEE-754 (2008)

## Accuracy for backward stable algorithms

- Accuracy of the computed sum $\leq (n-1) \times cond \times \mathbf{u}$
- $cond(\sum x_i) = \frac{\sum |x_i|}{|\sum x_i|}$
- No more significant digit in IEEE-b64 for large cond, *i.e.* $> 10^{16}$

## More accuracy ...

- More precision: double-double, quad-double, ...
- Compensated algorithms: Kahan(72), ..., Sum2(05), SumK(05)
- Accuracy of the computed sum $\lesssim \mathbf{u} + cond \times \mathbf{u}^K$

## ... but still depending on the conditioning

# Skip over the conditioning

## Distillation: iterate until faithful or exact rounding

- Error free transformation of $[x] \to [x^{(1)}] \to \cdots \to [x^*]$ such that $\sum x_i = \sum x_i^*$ and $[x^*]$ provides the expected rounded value.
- Kahan (87), ..., Zhu-Hayes: iFastSum (SISC-09)

## More space to keep everything

- Long accumulator, hardware oriented: Malcolm (71), Kulish (80)
- Cut the summands: AccSum (SISC-08), FastAccSum (SISC-09)
- Sum by fixed exponent: HybridSum (SISC-09), OnLineExact (TOMS-10)

## From faithful to exact rounding

- costly choice of the right side when closed to breakpoints
- *e.g.* $1 + 2^{-53} \pm 2^{-106}$

# Skip over the conditioning

Distillation: iterate until faithful or exact rounding

- Error free transformation of $[x] \rightarrow [x^{(1)}] \rightarrow \cdots \rightarrow [x^*]$ such that $\sum x_i = \sum x_i^*$ and $[x^*]$ provides the expected rounded value.
- Kahan (87), ..., Zhu-Hayes: iFastSum (SISC-09)

More space to keep everything

- Long accumulator, hardware oriented: Malcolm (71), Kulish (80)
- Cut the summands: AccSum (SISC-08), FastAccSum (SISC-09)
- Sum by fixed exponent: HybridSum (SISC-09), OnLineExact (TOMS-10)

From faithful to exact rounding

$\longrightarrow$ **Run-time and memory efficiencies are now the discriminant factors**

1 Why measure summation algorithm performance?

2 How to measure summation algorithm performance?

3 ILP and the PerPI Tool

4 Experiments with recent summation algorithms

5 Conclusion

# Reliable and significant measure of the time complexity?

## The classic way: count the number of flop

- A usual problem: double the accuracy of a computed result
- A usual answer for polynomial evaluation (degree $n$)

| Metric | Horner | CompHorner | DDHorner |
|---|---|---|---|
| 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 count vs. run-time measures

- Flop counts and measured run-times are not proportional
- Run-time measure is a very difficult experimental process
- Which one trust?

# 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 uncertainty of the hardware performance counters

- 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 and micro-architecture issues: multicore, hybrid, speculation
- Compiler options and its effects

# How to read the current literature?

Numerical results in S.M. Rump contributions (for summation)

- 26% for Sum2-SumK (SISC-05) : 9 pages over 34
- 20% for AccSum (SISC-08) : 7 pages over 35
- 20% for AccSumK-NearSum (SISC-08b) : 6 pages over 30
- **less that 3%** for FastAccSum (SISC-09) : 1 page over 37

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)*

...in the paper entitled *Ultimately Fast Accurate Summation*

# Software and System Performance experts' point of view

The limited *Accuracy of Performance Counter Measurements*

> *We caution performance analysts to be suspicious of cycle counts . . . gathered with performance counters.*
>
> *D. Zaparanuks, M. Jovic, M. Hauswirth (2009)*

*Can Hardware Performance Counters Produces Expected, Deterministic Results?*

> *In practice counters that should be deterministic show variation from run to run on the x86_64 architecture. . . . it is difficult to determine known "good" reference counts for comparison.*
>
> *V.M. Weaver, J. Dongarra (2010)*

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. Why measure summation algorithm performance?
2. How to measure summation algorithm performance?
3. ILP and the PerPI Tool
4. Experiments with recent summation algorithms
5. Conclusion

# ILP and the performance potential of an algorithm

Instruction Level Parallelism (ILP) describes the potential of the instructions of a program that can be executed simultaneously

Hennessy-Patterson's ideal machine (H-P IM)

- every instruction is executed one cycle after the execution one of the producers it depends
- no other constraint than the true instruction dependency (RAW)

Measure the #cycles and the #IPC running the code with the H-P IM

- maximal exploitation of the program ILP
- processor and ILP in practice: superscalar and out-of-order execution
- ILP measures the potential of the algorithm performance

# 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

Instruction and cycle counting

| | |
|---|---|
| | ... |
| 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

Cycle 0: i1 i2 i4

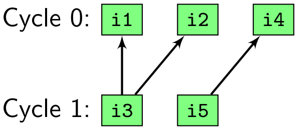
# What is ILP?

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



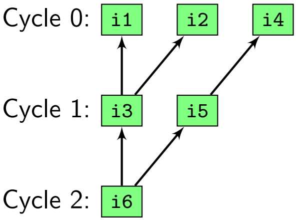
# What is ILP?

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



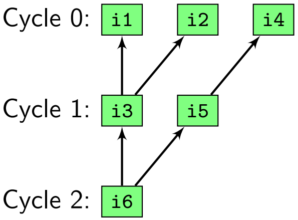
# What is ILP?

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




\# of instructions = 6, # of cycles = 3

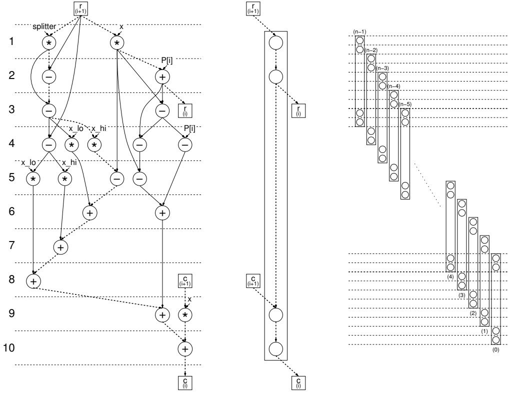
ILP = # of instructions/# of cycles = 2

# ILP explains why compensated algorithms run fast

N. Louvet, PhD (07)

CompHorner

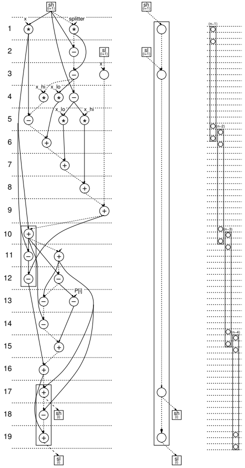
#C=2n+8, ILP≈ 11




(a) (b) (c)

DDHorner

#C=17n+2, ILP≈ 1.65




(a) (b) (c)

# The PerPI Tool automatizes this ILP analysis

PerPI: a pintool to analyse and visualise the ILP of x86-coded algorithms

- Pin (Intel) tool (http://www.pintool.org)
- Outputs: ILP measure (#C, #I), IPC histogram, data-dependency graph
- Input: x86_64 binary file
- Developed and maintained by B. Goossens and D. Parello (DALI)

1 Why measure summation algorithm performance?

2 How to measure summation algorithm performance?

3 ILP and the PerPI Tool

4 Experiments with recent summation algorithms

5 Conclusion

# Some recent accurate and fast summation algorithms

## Twice more precision

- Sum2: Compensated with a VectSum that uses TwoSum
- DDSum: Recursive sum + double-double arithmetic

## Faithful or exact rounding

- iFastSum: SumK with dynamic error control
- AccSum and FastAccSum: Adaptative computational effort wrt cond.
  Split the summands by chunk that sums exactly (width depends on $n$), careful sum of the chunks.
  Chunk cutting-line fixed in AccSum while more dynamic in FastAccSum
- HybridSum and OnLineExactSum: Exponent extraction of the summands, careful accumulation in one (HS) or two vectors (OLE) of fixed and short length (2048 in IEEE-b64), and distillate the (very for OLE) short vector with iFastSum.

# How to chose the data test?

Time complexity parameters of the summation algorithms

- Only $n$ for Sum2, SumK: constant accuracy improvement
- $n$ and *cond* for AccSum, iFastSum: adaptive accuracy improvement
- Exponent range of the summands: Z-H exhibit no influence for HybridSum and OnlineExactSum for large $n$
- Rump's generator of arbitrary ill-conditioned dot product (SISC-05), modified for summation and to cover an arbitrary exponent range.
- Length: $n \in [10^3, 10^7]$ and $cond \in [10^8, 10^{40}] = [\sqrt{1/\mathbf{u}}, 1/\mathbf{u}^{2.5}]$

# Our fuzzy PAPI picture

Parameters : sum length: $10^3$ to $10^6$, cond: $10^8$ to $10^{40}$

| cond | Sum | Sum2 | FastAccSum | iFastSum | HybridSum | OnLineExact |
|---|---|---|---|---|---|---|
| $10^8$ | 1 | 2–3 | 4–5 | 7–8 | 5 ($n > 10^5$) | 4 ($n > 10^5$) |
| $10^{16}$ | 1 | 2–3 | 5–6 | 7–8 | 5 ($n > 10^5$) | 4 ($n > 10^5$) |
| $10^{24}$ | 1 | – | 7 | 13 | 5 ($n > 10^5$) | 4 ($n > 10^5$) |
| $10^{32}$ | 1 | – | 8 | 18 | 5 ($n > 10^5$) | 4 ($n > 10^5$) |
| $10^{40}$ | 1 | – | 9 | 18+ | 5 ($n > 10^5$) | 4 ($n > 10^5$) |

Experimental process: PAPI, counter delay, hot caches, average over 50 samples for each $n$ and cond. ...

Intel(R) Core(TM) i7 CPU870 2.93GHz, x86_64, GNU/Linux noyau 2.6.38-8-generic

– gcc (4.6) -std=c99 -march=corei7 -mfpmath=sse -O3 -funroll-all-loops

– icc ($12.0.4_{20110427}$) -std=c99 -O3 -mtune=corei7 -xSSE -axsse4.2 -funroll-all-loops

# Focus inside OnLineExact

## Low level choices are crucial

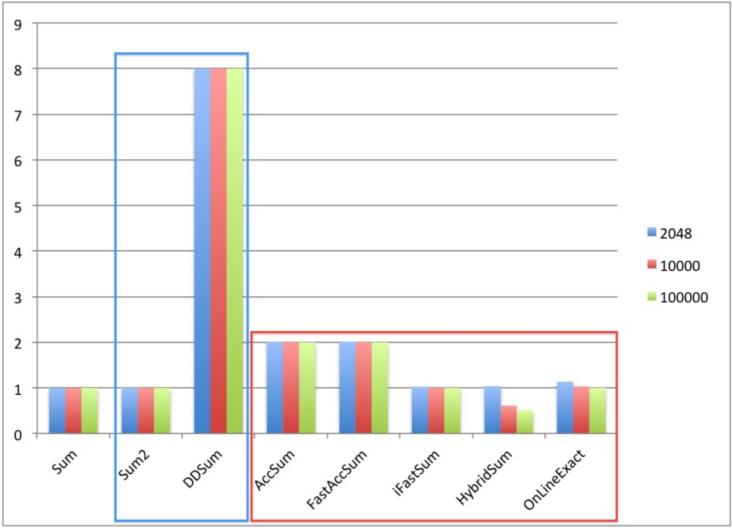
| cond $= 10^{16}$ | gcc | | icc | |
|---|---|---|---|---|
| n | 2sum | Fast2sum | 2sum | Fast2sum |
| $10^3$ | 13 | 11 | 13.7 | 9 |
| $10^4$ | 6.5 | 6.5 | 5.8 | 4 |
| $10^5$ | 5 | 5.5 | 3.8 | 3.3 |
| $10^6$ | 4.7 | 5.6 | 3.8 | 3.3 |

Cycle ratios (vs. Sum) vary for different EFT and compilers

# PerPI and reproducibility: one run is enough

```
start : <main>   (depth: 2, lcid: 104)
  stop  : <Sum>   (depth: 3, lcid: 10201)(cid: 10201)    I[13781]::C[10000]::ILP[1.3781]
  stop  : <Sum>   (depth: 3, lcid: 10203)(cid: 10203)    I[13781]::C[10000]::ILP[1.3781]
  stop  : <Sum>   (depth: 3, lcid: 10205)(cid: 10205)    I[13781]::C[10000]::ILP[1.3781]
  stop  : <iFastSumIn>   (depth: 3, lcid: 10207)(cid: 10207)    I[696088]::C[18043]::ILP[3
  stop  : <iFastSumIn>   (depth: 3, lcid: 10241)(cid: 10241)    I[696076]::C[18043]::ILP[3
  stop  : <iFastSumIn>   (depth: 3, lcid: 10275)(cid: 10275)    I[696076]::C[18043]::ILP[3
  start : <OnlineExactSum>   (depth: 3, lcid: 10309)
    stop  : <iFastSumIn>   (depth: 4, lcid: 10320)(cid: 10320)    I[29704]::C[611]::ILP[48
  stop  : <OnlineExactSum>   (depth: 3, lcid: 10309)(cid: 10309)    I[301467]::C[10607]::I
 stop  : <main>   (depth: 2, lcid: 104)(cid: 104)    I[2884900]::C[49320]::ILP[58.4935]
Global ILP (cid: 0)    I[2895541]::C[49572]::ILP[58.4108]
```

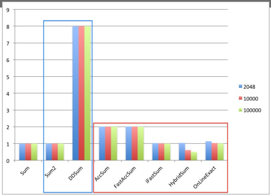
# PerPI: # cycle ratios for summation algorithms




Number of cycles: ratios vs. Sum

for *cond* $= 10^{32}$ and $n = 2048, 10^4, 10^5$

# PerPI: # cycle ratios for summation algorithms



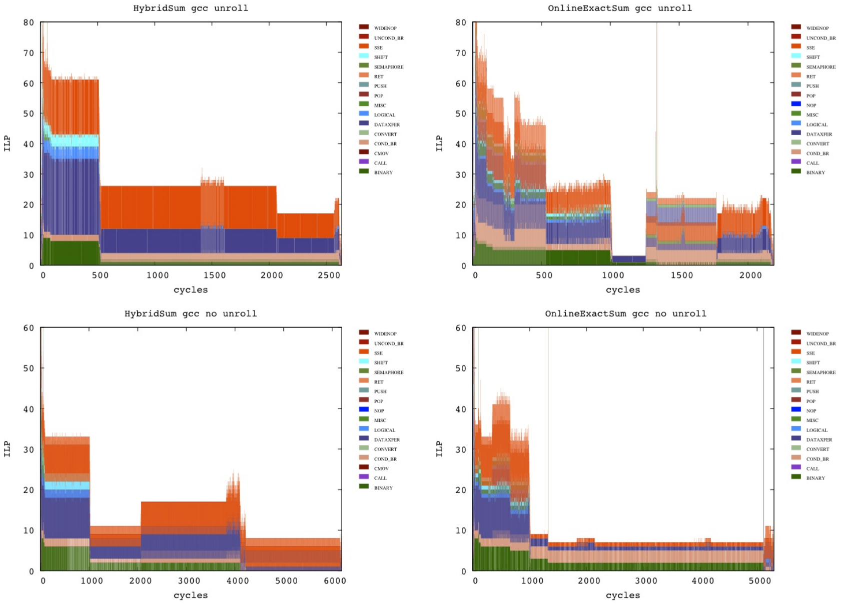

## Twice more accurate computed sum

- No overhead: compensation is the right choice

## Faithfully or exactly rounded computed sum

- The newest, the potentially fastest but be cautious: sensitive in practice
- FastAccSum (3n) not faster than AccSum (4n) [GLPP-Para10]
- OnLineExact for large $n$, else iFastSum
- PerPI highlights the control, *e.g.* iteration counters
- Less #C in HybridSum than in Sum?
  - Sum is unrolled 8 times by gcc but C forbids to change the evaluation order of the arithmetic expression
  - Every cycle of HybridSum has enough parallel work with different summands: 2 here
  - OnLineExact introduces dependency between iterations: $x[i]$ and $x[i+1]$ may have the same exponent

# HS (left) and OLE (right), unrolled (up) or not (down)



# Conclusion


1. Why measure summation algorithm performance?
2. How to measure summation algorithm performance?
3. ILP and the PerPI Tool
4. Experiments with recent summation algorithms
5. Conclusion

# Conclusion

- Highly accurate algorithm $\longrightarrow$ reliable performance evaluation
- Flop count: not significant
- Hardware counter based measure: uncertainty and no reproducibility
- PerPI: a software platform to analyze and visualise ILP
  - Reliable: reproducibility both in time and location
  - Realistic: correlation with measured ones
  - Useful: a detailed picture of the intrinsic behavior of the algorithm
  - Optimisation tool: analyse the effect of some hardware constraints [GLPP-Para10]
  - Exploratory tool: gives us the taste of the behavior of our algorithms running on "tomorrow" processors

# Conclusion

## Computing time: More science? Less hazard?

- No definitive answer
- PerPI result is far from perfect
  - Not abstract enough: instruction set dependence, compiler choice
  - Good abstraction level? Assembler program or high level programming language?

## Next step for f.p. summation: reproducibility to improve productivity

- Web site with common and shared resources: tested + test + make file sources, data files and generators, real and abstract associated measures
- Open and dynamic interaction: load your new algorithm, your new data, run them and let's contribute
- architectures? compilers?
- suggestions and partners are welcome!

# References I


D. H. Bailey.
Twelve ways to fool the masses when giving performance results on parallel computers.
*Supercomputing Review*, pages 54–55, Aug. 1991.

B. Goossens, P. Langlois, D. Parello, and E. Petit.
PerPI: A tool to measure instruction level parallelism.
In K. Jónasson, editor, *Applied Parallel and Scientific Computing - 10th International Conference, PARA 2010, Reykjavík, Iceland, June 6-9, 2010, Revised Selected Papers, Part I*, volume 7133 of *Lecture Notes in Computer Science*, pages 270–281. Springer, 2012.

N. J. Higham.
*Accuracy and Stability of Numerical Algorithms*.
SIAM, 2nd edition, 2002.

J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres.
*Handbook of Floating-Point Arithmetic*.
Birkhäuser Boston, 2010.

# References II


T. Ogita, S. M. Rump, and S. Oishi.
Accurate sum and dot product.
*SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005.

S. M. Rump.
Ultimately fast accurate summation.
*SIAM J. Sci. Comput.*, 31(5):3466–3502, 2009.

S. M. Rump, T. Ogita, and S. Oishi.
Accurate floating-point summation – part I: Faithful rounding.
*SIAM J. Sci. Comput.*, 31(1):189–224, 2008.

V. Weaver and J. Dongarra.
Can hardware performance counters produce expected, deterministic results?
In *3rd Workshop on Functionality of Hardware Performance Monitoring*, 2010.

D. Zaparanuks, M. Jovic, and M. Hauswirth.
Accuracy of performance counter measurements.
In *ISPASS*, pages 23–32. IEEE, 2009.

# References III


Y.-K. Zhu and W. B. Hayes.
Correct rounding and hybrid approach to exact floating-point summation.
*SIAM J. Sci. Comput.*, 31(4):2981–3001, 2009.

Y.-K. Zhu and W. B. Hayes.
Algorithm 908: Online exact summation of floating-point streams.
*ACM Transactions on Mathematical Software*, 37(3):37:1–37:13, Sept. 2010.