Performance Evaluation of Core Numerical Algorithms: a Tool to Measure Instruction Level Parallelism

Bernard Goossens, Philippe Langlois, David Parello, Éric Petit

University of Perpignan Via Domitia, France
DALI Research Project
Context and motivation

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

Floating point computation using IEEE-754 arithmetic

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

A classic answer:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Eval</th>
<th>AccEval1</th>
<th>AccEval2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flop count</td>
<td>2n</td>
<td>22n + 5</td>
<td>28n + 4</td>
</tr>
<tr>
<td>Flop count ratio</td>
<td>1</td>
<td>≈ 11</td>
<td>≈ 14</td>
</tr>
<tr>
<td>Measured #cycles ratio</td>
<td>1</td>
<td>2.8 – 3.2</td>
<td>8.7 – 9.7</td>
</tr>
</tbody>
</table>

Flop counts and running-times are not proportional. Why? Which one trust?
Average ratios for polynomials of degree 5 to 200
Working precision: IEEE-754 double precision

<table>
<thead>
<tr>
<th>System Description</th>
<th>Compiler</th>
<th>Comp Horner</th>
<th>DD Horner</th>
<th>DD Horner Comp Horner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4, 3.00 GHz (x87 fp unit)</td>
<td>GCC 4.1.2</td>
<td>2.8</td>
<td>8.5</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>ICC 9.1</td>
<td>2.7</td>
<td>9.0</td>
<td>3.4</td>
</tr>
<tr>
<td>Pentium 4, 3.00 GHz (sse2 fp unit)</td>
<td>GCC 4.1.2</td>
<td>3.0</td>
<td>8.9</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>ICC 9.1</td>
<td>3.2</td>
<td>9.7</td>
<td>3.4</td>
</tr>
<tr>
<td>Athlon 64, 2.00 GHz</td>
<td>GCC 4.1.2</td>
<td>3.2</td>
<td>8.7</td>
<td>3.0</td>
</tr>
<tr>
<td>Itanium 2, 1.4 GHz</td>
<td>GCC 4.1.1</td>
<td>2.9</td>
<td>7.0</td>
<td>2.4</td>
</tr>
<tr>
<td></td>
<td>ICC 9.1</td>
<td>1.5</td>
<td>5.9</td>
<td>3.9</td>
</tr>
</tbody>
</table>

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)*

Two separate jobs:

write and prove the algorithm vs. profile and tune one of its implementation

*We do not consider the source code (wrt the execution trace) since it’s not clear we understand it enough for tuning the execution analysis.*  
*M. Casas (Para 2010, last Sunday during the questions)*
Outline

1. How to choose the fastest algorithm?
2. The PerPI Tool
   - Goals and principles
3. The PerPI Tool: outputs and first examples
4. 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 wrt 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 $\rightarrow$ architecture, optimize algorithm $\rightarrow$ 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 (M. Gerndt, Monday)
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, DWP[ebp-16]</td>
<td>i1</td>
</tr>
<tr>
<td>mov edx, DWP[ebp-20]</td>
<td>i2</td>
</tr>
<tr>
<td>add edx, eax</td>
<td>i3</td>
</tr>
<tr>
<td>mov ebx, DWP[ebp-8]</td>
<td>i4</td>
</tr>
<tr>
<td>add ebx, DWP[ebp-12]</td>
<td>i5</td>
</tr>
<tr>
<td>add edx, ebx</td>
<td>i6</td>
</tr>
</tbody>
</table>

### Instruction and cycle counting

...
**What is ILP?**

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

---

### x86 binary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov eax, DWP[ebp-16]</code></td>
<td>i1</td>
</tr>
<tr>
<td><code>mov edx, DWP[ebp-20]</code></td>
<td>i2</td>
</tr>
<tr>
<td><code>add edx, eax</code></td>
<td>i3</td>
</tr>
<tr>
<td><code>mov ebx, DWP[ebp-8]</code></td>
<td>i4</td>
</tr>
<tr>
<td><code>add ebx, DWP[ebp-12]</code></td>
<td>i5</td>
</tr>
<tr>
<td><code>add edx, ebx</code></td>
<td>i6</td>
</tr>
</tbody>
</table>

---

### Instruction and cycle counting

**Cycle 0:** i1, i2, i4

---

9 / 20
What is ILP?

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

```
x86 binary
...

1: mov  eax, DWP[ebp-16]
2: mov  edx, DWP[ebp-20]
3: add  edx, eax
4: mov  ebx, DWP[ebp-8]
5: add  ebx, DWP[ebp-12]
6: add  edx, ebx

Instruction and cycle counting
Cycle 0: 1  2  4
Cycle 1: 3  5
```
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: \( i1, i2, i4 \)
- Cycle 1: \( i3, i5 \)
- Cycle 2: \( i6 \)
What is ILP?

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

### x86 binary

- \( \text{mov eax, DWP[ebp-16]} \)
- \( \text{mov edx, DWP[ebp-20]} \)
- \( \text{add edx, eax} \)
- \( \text{mov ebx, DWP[ebp-8]} \)
- \( \text{add ebx, DWP[ebp-12]} \)
- \( \text{add edx, ebx} \)

### Instruction and cycle counting

- **Cycle 0:**
  - \( i_1 \)
  - \( i_2 \)
  - \( i_4 \)

- **Cycle 1:**
  - \( i_3 \)
  - \( i_5 \)

- **Cycle 2:**
  - \( i_6 \)

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

\[
\text{ILP} = \frac{\# \text{ of instructions}}{\# \text{ of cycles}} = 2
\]
ILP explains why compensated algorithms are fast

ILP:

AccEval

\[ \approx 11 \]

AccEval2

\[ 1.65 \]
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 (M. Gerndt, Monday)

From ILP analysis to the PerPI tool

- 2008: prototype within a processor simulation platform (PPC asm)
- 2009: PerPI to analyse and visualise the ILP of x86-coded algorithms
  - Pintool (http://www.pintool.org)
  - Input: x86 binary file
  - Outputs: ILP measure, IPC histogram, data-dependency graph
What is PIN? Dynamic instrumentation


grep -t pintool -- ./a.out

\[pin\text{-}t\] pintool -- ./a.out

\text{PIN}

\text{x86 binary}

\begin{itemize}
  \item instr
  \item instr
  \item instr
  \item instr
  \item instr
  \item instr
  \item instr
  \item instr
  \item instr
\end{itemize}

\text{pintool}

\begin{itemize}
  \item instr
  \item instr
  \item instr
\end{itemize}

\text{hardware}
What is PIN? Dynamic instrumentation

$> \text{pin -t pintool \ -- \ ./a.out}$
What is PIN? Dynamic instrumentation

$> pin -t pintool -- ./a.out
What is PIN? Dynamic instrumentation

$> pin -t pintool -- ./a.out
What is PIN? Dynamic instrumentation

```bash
$> pin -t pintool -- ./a.out
```
What is PIN? Dynamic instrumentation

$>\text{pin -t pintool -- ./a.out}$
How to compute ILP?

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

---

**x86 binary**

```plaintext
... 

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 

... 
```

---
How to compute ILP?

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

### x86 binary

```
... 
call analyse 
mov eax, DWP[ebp-16] 
call analyse 
mov edx, DWP[ebp-20] 
call analyse 
add edx, eax 
call analyse 
mov ebx, DWP[ebp-8] 
call analyse 
add ebx, DWP[ebp-12] 
call analyse 
add edx, ebx 
... 
```
How to compute ILP?

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

**x86 binary**

```assembly
... 
call analyse 
mov eax,DWP[ebp-16] 
call analyse 
mov edx,DWP[ebp-20] 
call analyse 
add edx,eax 
call analyse 
mov ebx,DWP[ebp-8] 
call analyse 
mov ebx,DWP[ebp-12] 
call analyse 
add edx,ebx
... 
```

**Analyse routine**

<table>
<thead>
<tr>
<th>RC (register cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax : 0</td>
</tr>
<tr>
<td>ebx : 0</td>
</tr>
<tr>
<td>ecx : 0</td>
</tr>
<tr>
<td>edx : 0</td>
</tr>
<tr>
<td>ebx : 0</td>
</tr>
<tr>
<td>... : 0</td>
</tr>
<tr>
<td>ebp : 0</td>
</tr>
<tr>
<td>... : 0</td>
</tr>
</tbody>
</table>
How to compute ILP?

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

**x86 binary**

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

**Analyse routine**

- **input(i1): ebp**
- **cycle(i1)=max(RC[ebp]) = 0**
- **ni++; nc = max(nc, nc(i))**

**RC (register cycles)**

<table>
<thead>
<tr>
<th></th>
<th>eax</th>
<th>ebx</th>
<th>ecx</th>
<th>edx</th>
<th>ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>ni</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nc</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**input(i2): ebp**

- **cycle(i2)=max(RC[ebp]) = 0**
- **ni++; nc = max(nc, nc(i))**

**RC (register cycles)**

<table>
<thead>
<tr>
<th></th>
<th>eax</th>
<th>ebx</th>
<th>ecx</th>
<th>edx</th>
<th>ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>ni</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nc</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
How to compute ILP?

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

x86 binary

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

Analyze routine

```
input(i1): ebp
cycle(i1)=\text{max(RC[ebp])} = 0
output(i1): eax
new cycle(eax)=\text{cycle(i1)}+1 = 1
```

RC (register cycles)

```
eax : 1
ebx  : 0
ecx  : 0
dwx  : 0
ebx  : 0
... : 0
ebp  : 0
... : 0
```

ni = 1
nc = 0

\( \frac{13}{20} \)
How to compute ILP?

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

### x86 binary

```
... call analyse
mov  eax, DWP[ebp-16]
call analyse
mov  edx, DWP[ebp-20]
call analyse
add  edx, eax
    call analyse
mov  ebx, DWP[ebp-8]
call analyse
mov  ebx, DWP[ebp-12]
call analyse
add  edx, ebx
    call analyse
... add edx, ebx
... call analyse
```

### RC (register cycles)

```
ni = 1
nc = 0
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>1</td>
</tr>
<tr>
<td>ebx</td>
<td>0</td>
</tr>
<tr>
<td>ecx</td>
<td>0</td>
</tr>
<tr>
<td>edx</td>
<td>0</td>
</tr>
<tr>
<td>ebp</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>
**How to compute ILP?**

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

### x86 binary

```
... call analyse
mov eax,DWP[ebp-16]
call analyse
mov edx,DWP[ebp-20]
call analyse
add edx,eax i3
call analyse
mov ebx,DWP[ebp-8]
call analyse
add ebx,DWP[ebp-12]
call analyse
add edx,ebx i6
...```

### Analyse routine

- **input(i2): ebp**
- **cycle(i2) = max(RC[ebp]) = 0**
- **ni++; nc = max(nc, nc(i))**

### RC (register cycles)

```

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>1</td>
</tr>
<tr>
<td>ebx</td>
<td>0</td>
</tr>
<tr>
<td>ecx</td>
<td>0</td>
</tr>
<tr>
<td>edx</td>
<td>0</td>
</tr>
<tr>
<td>ebp</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>
```

\( ni = 2 \)

\( nc = 0 \)
How to compute ILP?

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

---

### x86 binary

```
... 
call analyse
mov eax, DWP[ebp-16]
call analyse
mov edx, DWP[ebp-20]
call analyse
add edx, eax
  
... 
```

### Analyse routine

| i1 | | i2 | | i3 | | i4 | | i5 | | i6 |
|----| |----| |----| |----| |----| |----| |----|

---

**input(i2): ebp**

**cycle(i2) = max(RC[ebp]) = 0**

**output(i2): edx**

**new cycle(edx) = cycle(i2) + 1 = 1**

---

**RC (register cycles)**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ebx</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ecx</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>edx</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ebx</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ebp</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A synthetic sample: \( e = (a+b) + (c+d) \)

**x86 binary**

```
... 
call analyse
mov eax,DWP[ebp-16]
call analyse
mov edx,DWP[ebp-20]
call analyse
add edx,eax
i3 
call analyse
mov ebx,DWP[ebp-8]
call analyse
mov ebx,DWP[ebp-12]
call analyse
add edx,ebx
i6 
call analyse
mov ebx,DWP[ebp-8]
call analyse
mov ebx,DWP[ebp-12]
call analyse
add edx,ebx
i6
... 
```

**Analyse routine**

```
i1
mov eax,DWP[ebp-16]
call analyse
mov edx,DWP[ebp-20]
call analyse
add edx,eax
i3
mov ebx,DWP[ebp-8]
call analyse
mov ebx,DWP[ebp-12]
call analyse
add edx,ebx
i6
```

**RC (register cycles)**

- \( ni = 2 \)
- \( nc = 0 \)

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>1</td>
</tr>
<tr>
<td>ebx</td>
<td>0</td>
</tr>
<tr>
<td>ecx</td>
<td>0</td>
</tr>
<tr>
<td>edx</td>
<td>1</td>
</tr>
<tr>
<td>ebp</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>
How to compute ILP?

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

**x86 binary**

```
... call analyse
mov eax,DWP[ebp-16]
call analyse
mov edx,DWP[ebp-20]
call analyse
add edx,eax
i3
... call analyse
mov ebx,DWP[ebp-8]
call analyse
add ebx,DWP[ebp-12]
call analyse
add edx,ebx
...  
```

**Analyse routine**

- **input(i3):** edx, eax
- **cycle(i3)=\max(\text{RC}[edx],\text{RC}[eax]) = 1**
- **ni++; nc = \max(nc, nc(i))**

**RC (register cycles)**

- \( \text{eax} : 1 \)
- \( \text{ebx} : 0 \)
- \( \text{ecx} : 0 \)
- \( \text{edx} : 1 \)
- \( \text{ebp} : 0 \)
- ... : 0

- \( \text{ni} = 3 \)
- \( \text{nc} = 1 \)
How to compute ILP?

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

Analyse routine

**x86 binary**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>call analyse</td>
</tr>
<tr>
<td></td>
<td>mov eax,DWP[ebp-16]</td>
</tr>
<tr>
<td></td>
<td>call analyse</td>
</tr>
<tr>
<td>i2</td>
<td>mov edx,DWP[ebp-20]</td>
</tr>
<tr>
<td></td>
<td>call analyse</td>
</tr>
<tr>
<td>i3</td>
<td>add edx,eax</td>
</tr>
<tr>
<td></td>
<td>call analyse</td>
</tr>
<tr>
<td>i4</td>
<td>mov ebx,DWP[ebp-8]</td>
</tr>
<tr>
<td></td>
<td>call analyse</td>
</tr>
<tr>
<td>i5</td>
<td>add ebx,DWP[ebp-12]</td>
</tr>
<tr>
<td></td>
<td>call analyse</td>
</tr>
<tr>
<td>i6</td>
<td>add edx,ebx</td>
</tr>
</tbody>
</table>
|    |   ...

**input(i3):** edx, eax

**cycle(i3)=max(RC[edx],RC[eax]) = 1**

**output(i3):** edx

**new cycle(edx)=cycle(i3)+1 = 2**

**RC (register cycles)**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>1</td>
</tr>
<tr>
<td>ebx</td>
<td>0</td>
</tr>
<tr>
<td>ecx</td>
<td>0</td>
</tr>
<tr>
<td>edx</td>
<td>2</td>
</tr>
<tr>
<td>ebp</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>

**ni = 3**

**nc = 1**
How to compute ILP?

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

**x86 binary**

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

**Analyse routine**

**RC (register cycles)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>1</td>
</tr>
<tr>
<td>ebx</td>
<td>0</td>
</tr>
<tr>
<td>ecx</td>
<td>0</td>
</tr>
<tr>
<td>edx</td>
<td>2</td>
</tr>
<tr>
<td>ebp</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>

\( ni = 3 \)
\( nc = 1 \)
How to compute ILP?

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

**x86 binary**

```
... 
call analyse 
mov eax, DWP[ebp-16] 
call analyse 
mov edx, DWP[ebp-20] 
call analyse 
add edx, eax 
call analyse 
mov ebx, DWP[ebp-8] 
call analyse 
add ebx, DWP[ebp-12] 
call analyse 
add edx, ebx 
... 
```

**Analyse routine**

```
input(i4): ebp 
cycle(i4)=\text{max}(RC[ebp]) = 0 
i++; nc = \text{max}(nc, nc(i))
```

**RC (register cycles)**

```
eax : 1 
ebx : 0 
cex : 0 
edx : 2 
ebx : 0 
... : 0 
ebp : 0 
... : 0
```

\( ni = 4 \)
\( nc = 1 \)
How to compute ILP?

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

Analyse routine

\[
\begin{align*}
\text{input}(i4) & : \text{ebp} \\
\text{cycle}(i4) & = \max(\text{RC}[\text{ebp}]) = 0 \\
\text{output}(i4) & : \text{ebx} \\
\text{new cycle}(\text{ebx}) & = \text{cycle}(i4) + 1 = 1
\end{align*}
\]

\( \text{ni} = 4 \)

\( \text{nc} = 1 \)

RC (register cycles)

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>1</td>
</tr>
<tr>
<td>ebx</td>
<td>1</td>
</tr>
<tr>
<td>ecx</td>
<td>0</td>
</tr>
<tr>
<td>edx</td>
<td>2</td>
</tr>
<tr>
<td>ebp</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>ebp</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>
How to compute ILP?

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

---

**x86 binary**

```assembly
... call analyse
mov eax,DWP[ebp-16]
call analyse
mov edx,DWP[ebp-20]
call analyse
add edx,eax
... call analyse
mov ebx,DWP[ebp-8]
call analyse
mov ebx,DWP[ebp-12]
call analyse
add edx,ebx
... call analyse
```

---

**Analyse routine**

**RC (register cycles)**

- \( ea : 1 \)
- \( ebx : 1 \)
- \( ecx : 0 \)
- \( edx : 2 \)
- \( ebp : 0 \)

\( ni = 4 \)

\( nc = 1 \)
How to compute ILP?

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

**x86 binary**

```
... 
call analyse 
mov eax, DWP[ebp-16] 
call analyse 
mov edx, DWP[ebp-20] 
call analyse 
add edx, eax 
call analyse 
mov ebx, DWP[ebp-8] 
call analyse 
add ebx, DWP[ebp-12] 
call analyse 
add edx, ebx 
... 
```

**Analysed routine**

```
input(i5): ebx, ebp 
cycle(i5) = \text{max}(RC[ebx], RC[ebp]) = 1 
```

```
input(i5): ebx, ebp 
cycle(i5) = \text{max}(RC[ebx], RC[ebp]) = 1 
```

```
i1++; nc = \text{max}(nc, nc(i)) 
```

**RC (register cycles)**

```
<table>
<thead>
<tr>
<th>Register</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>1</td>
</tr>
<tr>
<td>ebx</td>
<td>1</td>
</tr>
<tr>
<td>ecx</td>
<td>0</td>
</tr>
<tr>
<td>edx</td>
<td>2</td>
</tr>
<tr>
<td>ebp</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>
```

ni = 5  
nc = 2
How to compute ILP?

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

**Analyse routine**

**x86 binary**

\[
\begin{align*}
\text{call analyse} \\
\text{mov eax,DWP[ebp-16]} \\
\text{call analyse} \\
\text{mov edx,DWP[ebp-20]} \\
\text{call analyse} \\
\text{add edx,eax} \\
\text{call analyse} \\
\text{mov ebx,DWP[ebp-8]} \\
\text{call analyse} \\
\text{add ebx,DWP[ebp-12]} \\
\text{call analyse} \\
\text{add edx,ebx} \\
\end{align*}
\]

**input(i5): ebx, ebp**

\( \text{cycle(i5)} = \text{max}(\text{RC}[ebx],\text{RC}[ebp]) = 1 \)

**output(i5): ebx**

\( \text{new cycle(ebx)} = \text{cycle(i5)} + 1 = 2 \)

**RC (register cycles)**

\[
\begin{array}{c}
\text{eax} : 1 \\
\text{ebx} : 2 \\
\text{ecx} : 0 \\
\text{edx} : 2 \\
\text{ebx} : 0 \\
\text{...} : 0 \\
\text{ebp} : 0 \\
\text{...} : 0 \\
\end{array}
\]

\( ni = 5 \)

\( nc = 2 \)
How to compute ILP?

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

### x86 binary

```assembly
... call analyse
mov eax,DWP[ebp-16]
call analyse
mov edx,DWP[ebp-20]
call analyse
add edx,eax
i3 call analyse
mov ebx,DWP[ebp-8]
call analyse
mov ebx,DWP[ebp-12]
call analyse
add edx,ebx
i6 ...
```

### Analyse routine

#### RC (register cycles)

- \( ni = 5 \)
- \( nc = 2 \)

- eax : 1
- ebx : 2
- ecx : 0
- edx : 2
- ebp : 0
- ... : 0
How to compute ILP?

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

Analyse routine

input(i6): edx, ebx

$\text{cycle}(i6) = \max(\text{RC}[edx], \text{RC}[ebx]) = 2$

$\text{ni} = 6$
$\text{nc} = 2$

$\text{ni} + 1; \text{nc} = \max(\text{nc}, \text{nc}(i))$

RC (register cycles)

- eax : 1
- ebx : 2
- ecx : 0
- edx : 2
- ebp : 0
- ... : 0
How to compute ILP?

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

**x86 binary**

```
... 
call analyse 
mov eax,DWP[ebp-16] 
call analyse 
mov edx,DWP[ebp-20] 
call analyse 
add edx,eax 
call analyse 
mov ebx,DWP[ebp-8] 
call analyse 
add ebx,DWP[ebp-12] 
call analyse 
add edx,ebx 
... 
```

**Analyse routine**

```
input(i6): edx, ebx 
cycle(i6)=\max(\text{RC}[edx],\text{RC}[ebx]) = 2 
output(i6): edx 
new cycle(edx)=cycle(i6)+1 = 3 
```

**RC (register cycles)**

<table>
<thead>
<tr>
<th></th>
<th>eax</th>
<th>edx</th>
<th>ecx</th>
<th>edx</th>
<th>ebx</th>
<th>...</th>
<th>ebp</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\( ni = 6 \) 
\( nc = 2 \)

13 / 20
How to compute ILP?

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

Analyse routine

RC (register cycles)

\[
\begin{array}{c}
\text{ni} = 6 \\
\text{nc} = 2 \\
\end{array}
\]

\[
\begin{array}{c}
eax : 1 \\
edx : 3 \\
ecx : 0 \\
edx : 2 \\
ebx : 0 \\
\ldots : 0 \\
ebp : 0 \\
\ldots : 0 \\
\end{array}
\]
How to compute ILP?

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

```
x86 binary
...
call analyse
mov eax,DWP[ebp-16]
call analyse
mov edx,DWP[ebp-20]
call analyse
add edx,eax
i3
call analyse
mov ebx,DWP[ebp-8]
call analyse
add ebx,DWP[ebp-12]
call analyse
add edx,ebx
i6...
```

Analyse routine

RC (register cycles)

- ni = 6
- nc = 2
- ILP = 6/3 = 2
Outline

1. How to choose the fastest algorithm?
2. The PerPI Tool
3. The PerPI Tool: outputs and first examples
4. Conclusion
Simulation produces reproducible results

```plaintext
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)  I[13]:C[9]:ILP[1.44444]
    stop : call_gmon_start  (depth: 4 rtn_s_d: 0)
    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[1.83333]
    stop :  _init  (depth: 3 rtn_s_d: 0)  I[41]:C[26]:ILP[1.57692]
    start : __libc_csu_init  (depth: 2 rtn_s_d: 0)  I[63]:C[39]:ILP[1.61538]
    stop :  main  (depth: 2 rtn_s_d: 0)
    start :  Horner  (depth: 3 rtn_s_d: 0)  I[519]:C[206]:ILP[2.51942]
    stop :  Horner  (depth: 3 rtn_s_d: 0)
    start :  CompHorner  (depth: 3 rtn_s_d: 0)  I[3732]:C[318]:ILP[11.7358]
    stop :  CompHorner  (depth: 3 rtn_s_d: 0)
    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
Visualisation of the instruction dependence graph
New FastAccSum is announced to be faster than AccSum:

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

- 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
- Let’s exploit it!
Instruction dependence graph to compare two algorithms

- 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.
1. How to choose the fastest algorithm?
2. The PerPI Tool
3. The PerPI Tool: outputs and first examples
4. 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
- 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
The key to performance is to understand the algorithm and the architecture interaction.

Fred Gustavson, Para 2010, last Monday.
Bo Kågström, Para 2010, last Tuesday.