Computer Assisted Proofs – tools, methods and applications

Dagstuhl Seminar, 16-20 november 2009, Germany

Processor simulation: a new way for the performance analysis of numerical algorithms

Bernard Goossens, Philippe Langlois, David Parello

University of Perpignan Via Domitia

DALI Research Project

UPVD

Université de Perpignan Via Domitia

DALI

Digits, Architectures et Logiciels Informatiques

Acknowledgement

Participation at this seminar is granted by
EVA-Flo: Evaluation and Automatic Validation for Floating-Point Computing.
ANR funded project 2007-2010. Chair: Nathalie Revol.

Context and connection with computer-assisted proofs

Floating point computation using IEEE-754 arithmetic

Improving and validating the accuracy of some numerical algorithms thanks to error-free transformations

- Summation and dot product (Ogita-Rump-Oishi:2005), polynomial evaluation with Horner algorithm (Graillat-PhL-Louvet in JJIAM, to appear), triangular linear system resolution (PhL-Louvet:2008)

These algorithms are fast in terms of measured computing time

- Faster than other existing solutions: double-double, quad-double libraries
  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?

Outline

These algorithms are fast in terms of measured times ???

		$\frac{\text{CompHorner}}{\text{Horner}}$	$\frac{\text{DDHorner}}{\text{Horner}}$	$\frac{\text{DDHorner}}{\text{CompHorner}}$
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

Parameters[1] are numerous and most of them have a short expected time life

- processor (family, unit, frequency), compiler (family, version, options)

Conclusion: CompHorner runs a least two times faster than DDHorner

- it's not a theorem
- it's only a blurred picture

[1] Average ratios for polynomials of degree 5 to 200; wp = IEEE-754 double precision

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 process scheduling, interrupts
- non deterministic dynamic instruction scheduler or branch predictor, initial state of the branch predictor
- external conditions: temperature of the room
- accuracy of the timings: no constant cycle period on modern processors (Intel core, i7, ...)

Uncertainty increases as computer system complexity does

- multicore architectures

How to explain that the flop count and running time measures are not proportional?

	Horner	CompHorner	DDHorner	$\frac{\text{DDHorner}}{\text{CompHorner}}$
Flop count	2n	**22n + 5**	**28n + 4**	≈ 1.3
Flop ratio	1	≈ 11	≈ 14	
Measured #cycles ratio	1	2.8 – 3.2	8.7 – 9.7	3 – 3.5

How to explain that the flop count and running time measures are not proportional?

	Horner	CompHorner	DDHorner	$\frac{\text{DDHorner}}{\text{CompHorner}}$
Flop count	2n	$\mathbf{22n + 5}$	$\mathbf{28n + 4}$	≈ 1.3
Flop ratio	1	≈ 11	≈ 14	
Measured #cycles ratio	1	2.8 – 3.2	8.7 – 9.7	3 – 3.5

Previous results

- First question opened by T. Ogita, S.M. Rump and S. Oishi (SISC, 2005).
- SCAN 2006 (Duisburg): very first results about instruction level parallelism (ILP)
- Nicolas Louvet's PhD (nov. 2007), research report (LaLo08)

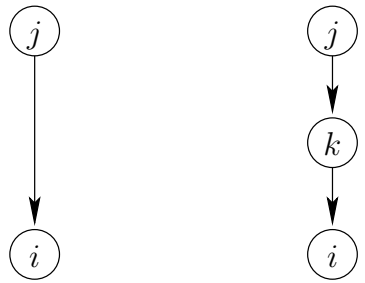
Outline

Instruction-level parallelism (ILP)

Instruction-level parallelism (ILP) represents the potential of the instructions of a program that can be executed simultaneously.

An instruction i is data dependent on an instruction j if either

- instruction j produces a result that may be used by instruction i,
- their exists a chain of dependencies of the first type between i and j.



If two instructions are data dependent, they cannot execute simultaneously

- True data dependence (RAW) *vs.* false data dependence (WAR and WAW): dynamic register renaming suppresses false dependencies.

Dependencies are properties of the algorithm: the presence of a data dependence in an instruction sequence reflects a data dependence in the source code.

How processors are designed to exploit ILP?

For a given instruction set, let the compiler or the processor break the instruction sequence into an overlapping instruction flow.

- Micro-architectural techniques: pipelining, superscalar execution, fetching blocks of instructions, dynamic branch prediction, dynamic scheduling with register renaming, out-of-order execution, load address speculation, ...
- Static or dynamic implementation of these techniques

  Itanium *vs.* x86-32, x86-64, PPC, ...

ILP: some key points

Pipelining: multiple instructions partially overlap in execution (IPC ≤ 1)

Superscalar execution: multiple execution pipelined units execute multiple instructions in parallel (IPC > 1)

		clock cycle number						
inst. number	inst. type	1	2	3	4	5	6	7
i	Int	IF	ID	EX	MEM	WB		
$i+1$	Fl	IF	ID	EX	MEM	WB		
$i+2$	Int		IF	ID	EX	MEM	WB	
$i+3$	Fl		IF	ID	EX	MEM	WB	
$i+4$	Int			IF	ID	EX	MEM	WB
$i+5$	Fl			IF	ID	EX	MEM	WB

Out-of-order execution just satisfying the data dependencies

How to quantify instruction-level parallelism in a program?

Step 1: let us consider an ideal processor (Hennessy-Patterson)

The ideal processor runs the program such that

an instruction is scheduled on the cycle immediately following the execution of the predecessor on which it depends

- all but true data dependencies are removed
- conditional branches/loops are perfectly predicted
- memory accesses are also perfect
- name dependencies are removed
- an unlimited number of instructions can be executed in the same cycle
- every instruction is executed in one clock cycle

How to quantify instruction-level parallelism in a program?

Step 1: let us consider an ideal processor (Hennessy-Patterson)

The ideal processor runs the program such that

an instruction is scheduled on the cycle immediately following
the execution of the predecessor on which it depends

Step 2: compute its optimal IPC = IPC on the ideal processor

$$\begin{aligned} \text{IPC} &= \text{average number of instructions executed in one clock cycle} \\ &= \frac{\text{Total number of instructions}}{\text{Total latency of the program}} \end{aligned}$$

How to quantify instruction-level parallelism in a program?

Step 1: let us consider an ideal processor (Hennessy-Patterson)

The ideal processor runs the program such that

an instruction is scheduled on the cycle immediately following the execution of the predecessor on which it depends

Step 2: compute its optimal IPC = IPC on the ideal processor

$$\begin{aligned} \text{IPC} &= \text{average number of instructions executed in one clock cycle} \\ &= \frac{\text{Total number of instructions}}{\text{Total latency of the program}} \end{aligned}$$

Conclusion

More IPC implies better performances on modern processors.

Outline

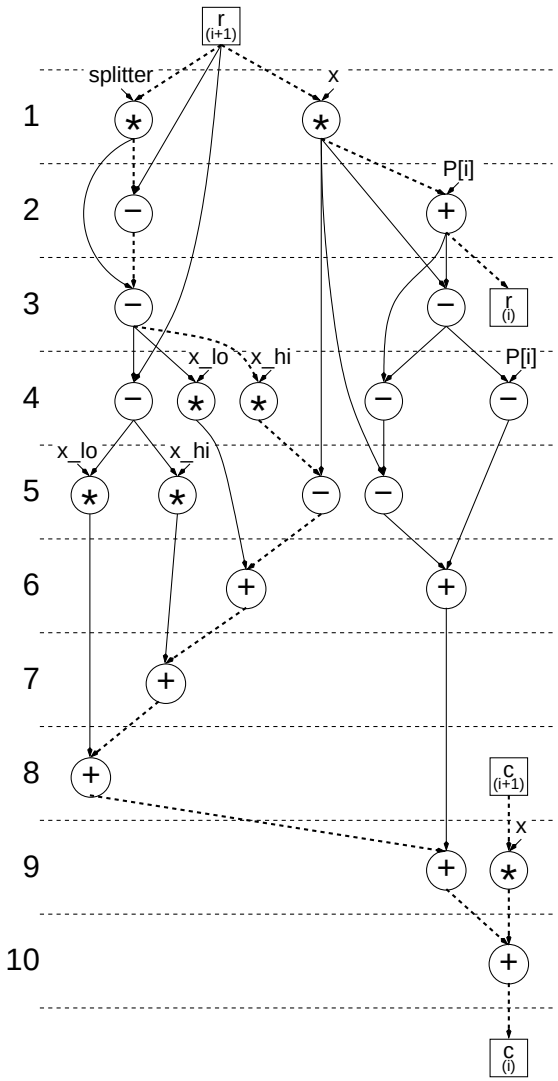
Optimal IPC of CompHorner: the manual analysis

C implementation of CompHorner

```
double CompHorner(double *P, int n, double x) {
  double p, r, c, pi, sig;
  double x_hi, x_lo, hi, lo, t;
  int i;
  /* Split(x_hi, x_lo, x) */
  t = x * _splitter_;
  x_hi = t - (t - x); x_lo = x - x_hi;
  r = P[n]; c = 0.0;
  for(i=n-1; i>=0; i--) {
    /* TwoProd(p, pi, s, x); */
    p = r * x;
    t = r * _splitter_;
    hi = t - (t - r);
    lo = r - hi;
    pi = (((hi*x_hi - p) + hi*x_lo)
          + lo*x_hi) + lo*x_lo;
    /* TwoSum(s, sigma, p, P[i]); */
    r = p + P[i];
    t = r - p;
    sig = (p - (r - t)) + (P[i] - t);
    /* Computation of the error term */
    c = c * x + (pi+sig);
  }
  return(r+c);
}
```

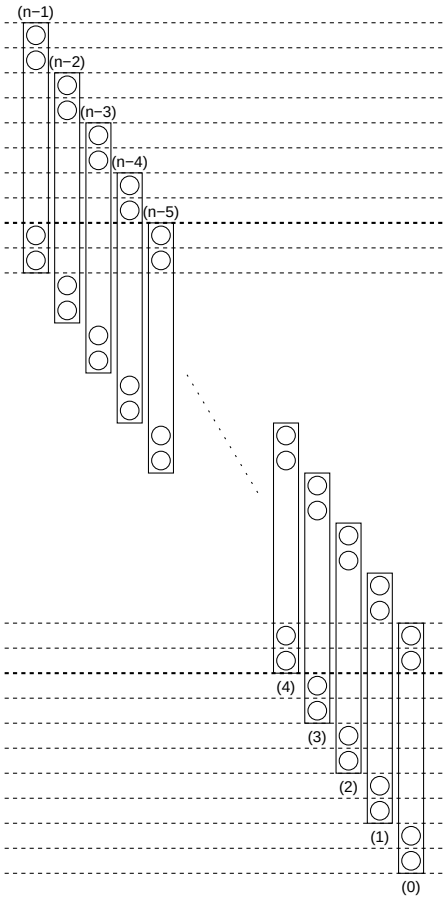
1. Counting the number of flop in the algorithm
   - $22n + 5$ flop in the accurate evaluation of a degree n polynomial

Optimal IPC of CompHorner: the manual analysis



1. Counting the number of flop in the algorithm
   - $22n + 5$ flop in the accurate evaluation of a degree n polynomial
2. From within the data flow graph, find the execution latency for one iteration
   - the latency of one iteration is 10 cycles.

Optimal IPC of CompHorner: the manual analysis



1. Counting the number of flop in the algorithm
   - $22n + 5$ flop in the accurate evaluation of a degree n polynomial
2. From within the data flow graph, find the execution latency for one iteration
   - the latency of one iteration is 10 cycles.
3. Find the total execution latency within the data flow graph
   - consecutive iterations overlap by 8 cycles

Optimal IPC of CompHorner: the manual analysis

C implementation of CompHorner

```
double CompHorner(double *P, int n, double x) {
  double p, r, c, pi, sig;
  double x_hi, x_lo, hi, lo, t;
  int i;
  /* Split(x_hi, x_lo, x) */
  t = x * _splitter_;
  x_hi = t - (t - x); x_lo = x - x_hi;
  r = P[n]; c = 0.0;
  for(i=n-1; i>=0; i--) {
    /* TwoProd(p, pi, s, x); */
    p = r * x;
    t = r * _splitter_;
    hi = t - (t - r);
    lo = r - hi;
    pi = (((hi*x_hi - p) + hi*x_lo)
          + lo*x_hi) + lo*x_lo;
    /* TwoSum(s, sigma, p, P[i]); */
    r = p + P[i];
    t = r - p;
    sig = (p - (r - t)) + (P[i] - t);
    /* Computation of the error term */
    c = c * x + (pi+sig);
  }
  return(r+c);
}
```

1. Counting the number of flop in the algorithm
   - $22n + 5$ flop in the accurate evaluation of a degree n polynomial
2. From within the data flow graph, find the execution latency for one iteration
   - the latency of one iteration is 10 cycles.
3. Find the total execution latency within the data flow graph
   - consecutive iterations overlap by 8 cycles
4. Optimal IPC for CompHorner

$$\text{IPC}_{\text{CompHorner}} = \frac{22n+5}{2n+8} \approx 11 \text{ instructions per cycle.}$$

Measured performances *vs.* potential of performances

	Horner	CompHorner	DDHorner	$\frac{\text{DDHorner}}{\text{CompHorner}}$
Flop count	2n	**22n + 5**	**28n + 4**	≈ 1.3
Optimal IPC	1	11	1.6	≈ 6.8
Optimal #cycles: avg.	2n	2n	17.5n	
Optimal #cycles: ratio	1	≈ 1	≈ 8.75	≈ 8.75
Measured #cycles: ratio	1	2.7 – 3.2	8.5 – 9.7	2.9 – 3.3

More ILP available in CompHorner than in DDHorner,

so better practical performances on modern processors

Observing that on the ideal processor,

optimal average number of cycles = Flop count / Optimal IPC

- CompHorner run as fast as Horner
- CompHorner run about 8 times faster than DDHorner

We need processors exploiting more ILP to verify this performance potential

Outline

Simulation of processor

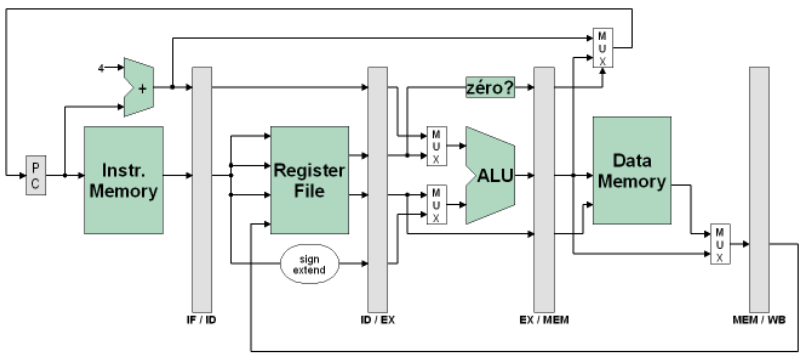
Software simulation is classic for processor design

- complexity, costly, one step final process.
- SimpleScalar, System C, Liberty, Asim (Intel ppty), MaxCore (ARM ppty)

Unisim: INRIA + CEA + Princeton U. + HiPEAC (IST Network of Excellence on High-Performance Embedded Architectures and Compilers = 10 EEC countries + STMicroelectronics, ARM, Philips, Infineon)

- Library of processors (PowerPC, ARM), of processor component: processor, memory (main, caches), network (buses)
- Open simulation framework
- Cycle-level simulators are cycle accurate models characterized by an high accuracy on performance evaluation comparing to the real hardware.

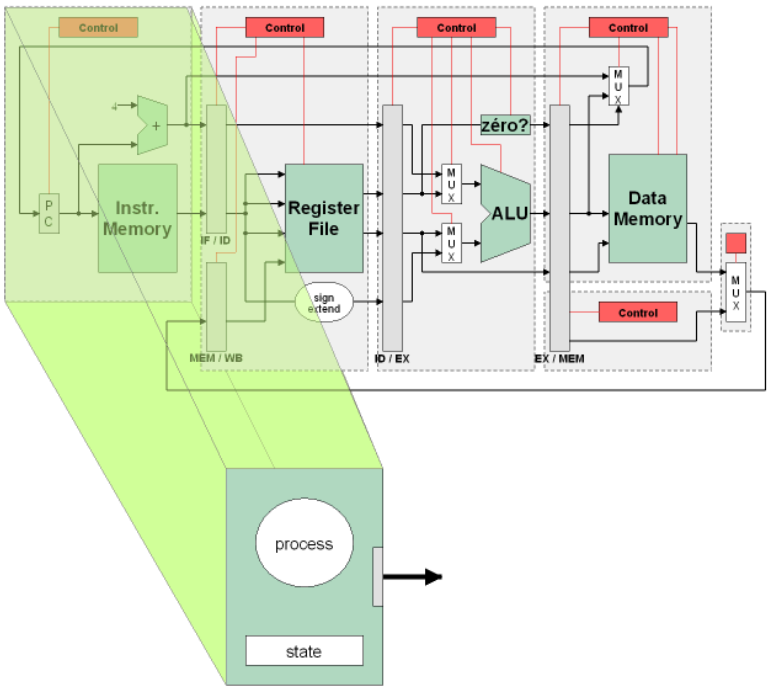
Very basic principles of UniSim



Starting with a block diagram

- Main hardware components of a 5-stage pipelined DLX

Very basic principles of UniSim

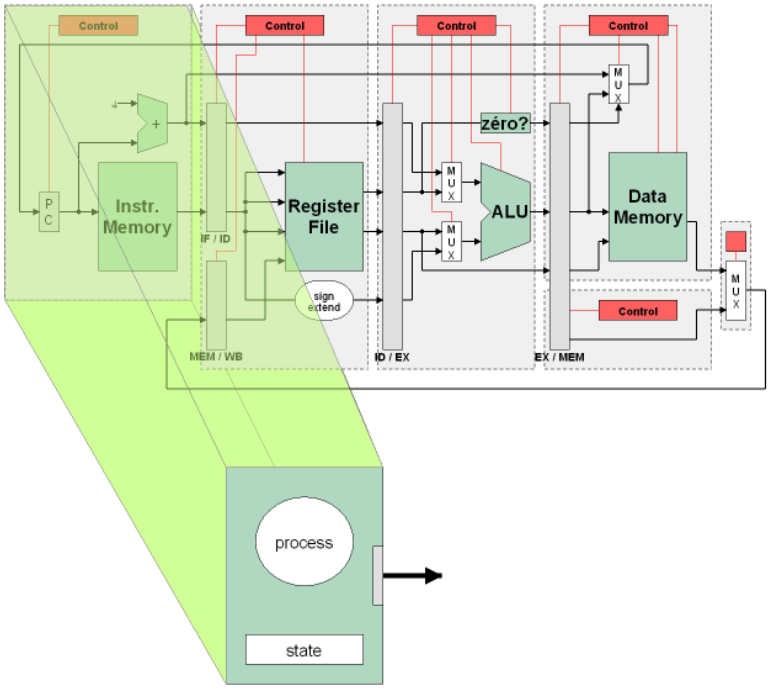


Starting with a block diagram

- Main hardware components of a 5-stage pipelined DLX

One component = One module

Very basic principles of UniSim



Starting with a block diagram

- Main hardware components of a 5-stage pipelined DLX

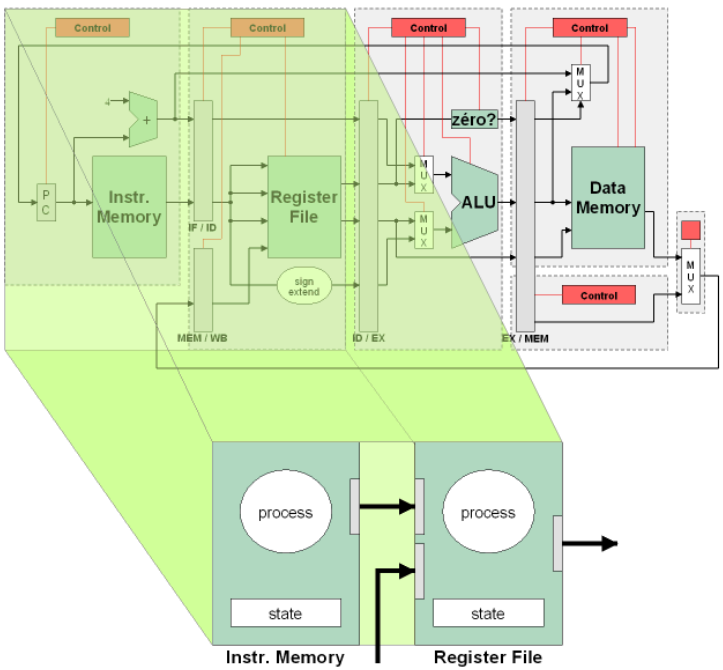
One component = One module
= One C++ object

- its processes: C++ methods
- its state: C++ class properties

```
class InstructionMemory : public module
{
    uint32_t mem[65536];
    uint32_t pc;
    uint32_t instr;

    void fetch()
    {
        instr = mem[pc];
        pc = pc + 4;
    }
}
```

Very basic principles of UniSim



Starting with a block diagram

- Main hardware components of a 5-stage pipelined DLX

One component = One module
= One C++ object

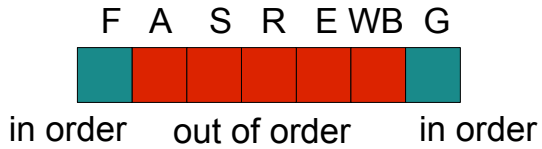
- its processes: C++ methods
- its state: C++ class properties

Port-based communication between modules + a clock

Our ideal processor simulation

OoOSim

- a simulated Power PC with an arbitrary superscalar rate
- RISC instruction set, 7-stage pipeline, out-of-order execution



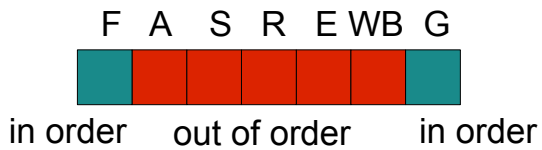
How to simulate the ideal processor?

- F: the branch prediction is perfect (emulation of the true result)
- A: the register renaming step avoids all but true dependencies
- S: the instruction queue is arbitrarily large (to encompass poor ILP cases)
- R: the length of the register file is arbitrarily large ("infinite" number of physical registers)
- E: the number of execution units of every type (iop, flop,) at least equals the superscalar degree
- WB: the number of WB at least equals the superscalar degree
- G: the length of the ROB queue is arbitrarily large and the number of retired instructions at least equals the superscalar degree

Our ideal processor simulation

OoOSim

- a simulated Power PC with an arbitrary superscalar rate
- RISC instruction set, 7-stage pipeline, out-of-order execution



How to simulate the ideal processor?

The experimental process

- We increase the superscalar rate at least until the IPC becomes constant
- In practice, superscalar rates 1, 2, 4, 16 and 32 have been used
- Every program is compiled with option
  -O3 -finline-functions -Wall -mno-fused-madd

A picture of the processor state at every cycle

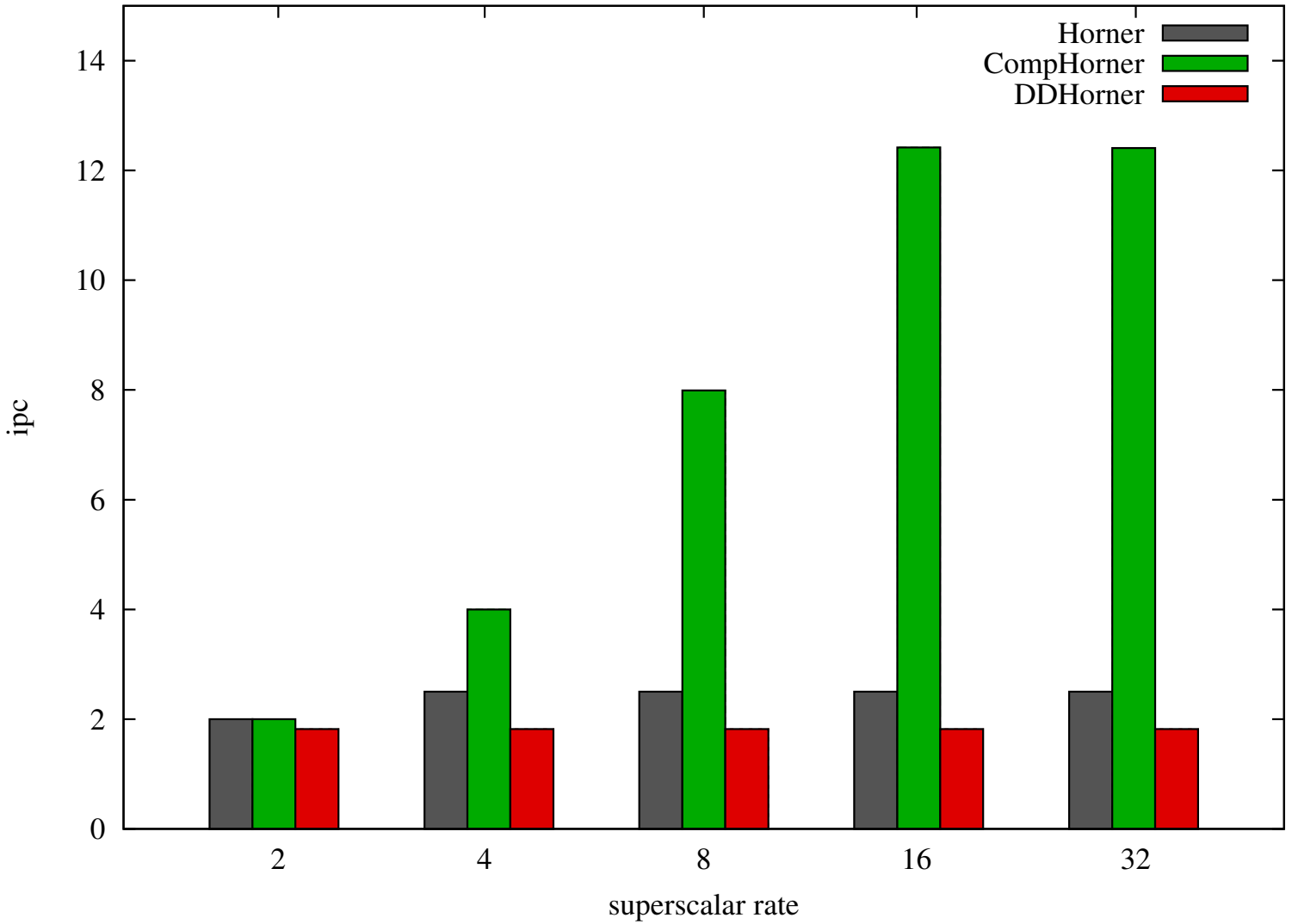
100088a0	sc	f: 8969	a: 8977	s: 9237	r: 9238	e: 9239	w: 9240	r: 9243
100088a4	mfcr r0	f: 9243	a: 9244	s: 9245	r: 9246	e: 9247	w: 9248	r: 9251
100088a8	mr r31, r3	f: 9243	a: 9244	s: 9245	r: 9246	e: 9247	w: 9248	r: 9251
100088ac	andis. r9, r0, 4096	f: 9243	a: 9244	s: 9245	r: 9248	e: 9249	w: 9250	r: 9253
100088b0	bc 13, 2, 0x100088bc	f: 9243	a: 9244	s: 9245	r: 9250	e: 9251	w: 9252	r: 9255
100088b4	bl 0x10001a6c	f: 9243	a: 9244	s: 9245	r: 9246	e: 9247	w: 9248	r: 9255
10001a6c	lis r9, 4104	f: 9243	a: 9244	s: 9245	r: 9246	e: 9247	w: 9248	r: 9255
10001a70	lwz r0, 3540(r9)	f: 9243	a: 9244	s: 9245	r: 9248	e: 9249	w: 9252	r: 9255
10001a74	lis r9, 4104	f: 9243	a: 9244	s: 9245	r: 9246	e: 9247	w: 9248	r: 9255
10001a78	addi r3, r9, 3668	f: 9243	a: 9245	s: 9246	r: 9248	e: 9249	w: 9250	r: 9255
10001a7c	cmpi cr0, 0, r0, 0	f: 9243	a: 9245	s: 9246	r: 9252	e: 9253	w: 9254	r: 9257
10001a80	bclr 13, 2	f: 9243	a: 9245	s: 9246	r: 9254	e: 9255	w: 9256	r: 9259
100088b8	stw r31, 0(r3)	f: 9243	a: 9245	s: 9246	r: 9250	e: 9251	w: 9252	r: 9259
100088bc	li r0, 1	f: 9243	a: 9245	s: 9246	r: 9247	e: 9248	w: 9249	r: 9259
100088c0	mr r3, r30	f: 9243	a: 9245	s: 9246	r: 9247	e: 9248	w: 9249	r: 9259

A picture of the processor state at every cycle

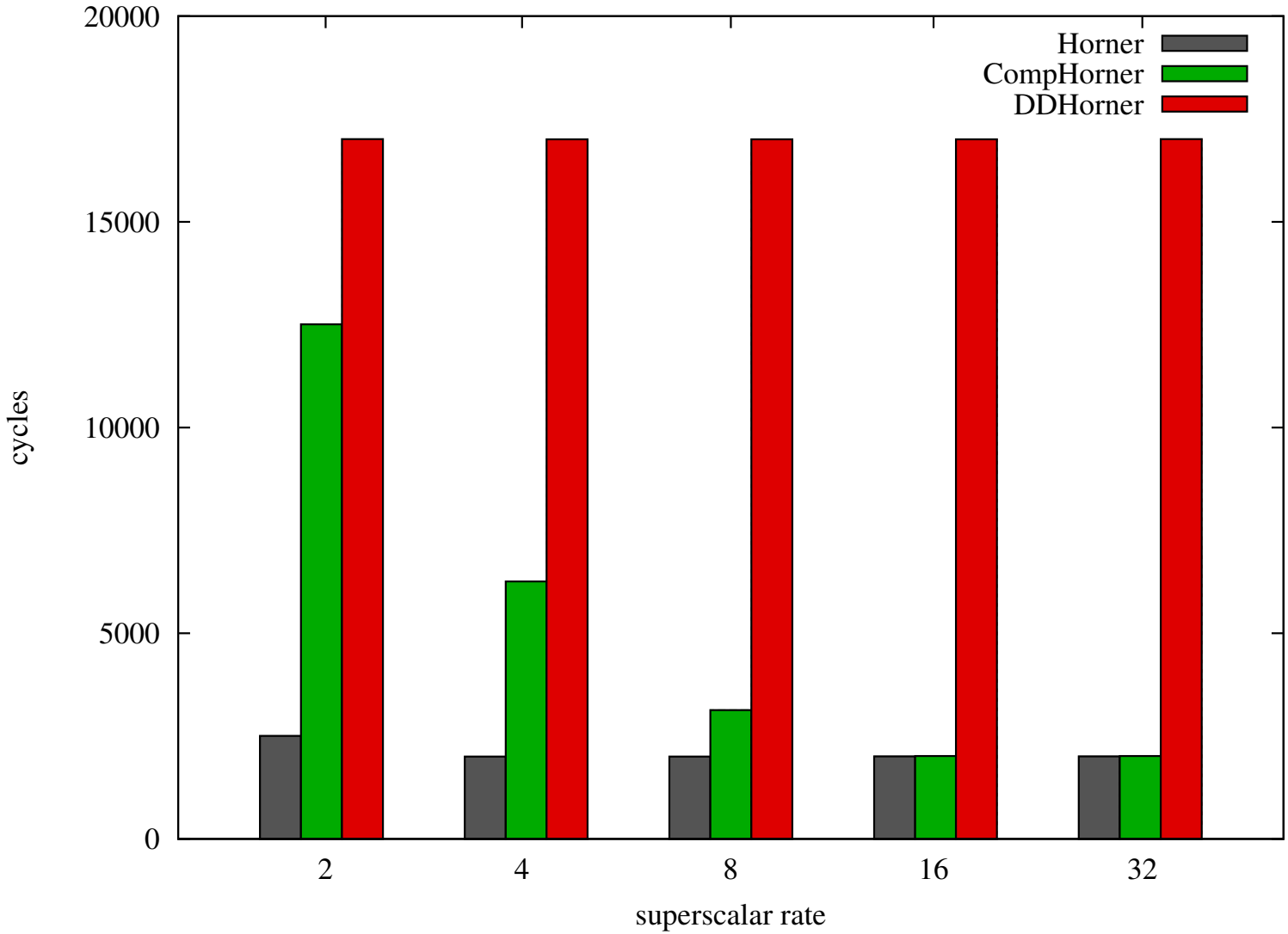
Feuille1

RetIns: 34652	sim_num_ins: 34556	sim_num_refs: 6212	sim_num_loads: 3544	stall_sched: 3189
RetIns: 34653	sim_num_ins: 34557	sim_num_refs: 6212	sim_num_loads: 3544	stall_sched: 3189
RetIns: 34654	sim_num_ins: 34558	sim_num_refs: 6212	sim_num_loads: 3544	stall_sched: 3189
RetIns: 34655	sim_num_ins: 34559	sim_num_refs: 6212	sim_num_loads: 3544	stall_sched: 3189
RetIns: 34656	sim_num_ins: 34560	sim_num_refs: 6212	sim_num_loads: 3544	stall_sched: 3189
RetIns: 34657	sim_num_ins: 34561	sim_num_refs: 6212	sim_num_loads: 3544	stall_sched: 3189
RetIns: 34658	sim_num_ins: 34562	sim_num_refs: 6212	sim_num_loads: 3544	stall_sched: 3189
RetIns: 34659	sim_num_ins: 34563	sim_num_refs: 6213	sim_num_loads: 3545	stall_sched: 3189
RetIns: 34660	sim_num_ins: 34564	sim_num_refs: 6213	sim_num_loads: 3545	stall_sched: 3189
RetIns: 34661	sim_num_ins: 34565	sim_num_refs: 6213	sim_num_loads: 3545	stall_sched: 3189
RetIns: 34662	sim_num_ins: 34566	sim_num_refs: 6213	sim_num_loads: 3545	stall_sched: 3189
RetIns: 34663	sim_num_ins: 34567	sim_num_refs: 6213	sim_num_loads: 3545	stall_sched: 3189
RetIns: 34664	sim_num_ins: 34568	sim_num_refs: 6214	sim_num_loads: 3545	stall_sched: 3189
RetIns: 34665	sim_num_ins: 34569	sim_num_refs: 6214	sim_num_loads: 3545	stall_sched: 3189
RetIns: 34666	sim_num_ins: 34570	sim_num_refs: 6214	sim_num_loads: 3545	stall_sched: 3189

The simulated ideal proc runs accurate Horner algorithms



The simulated ideal proc runs accurate Horner algorithms



Simulation results *vs.* current processors experiments
Simulated IPC *vs.* theoretical IPC and consequences

	Horner	CompHorner	DDHorner	$\frac{\text{DDHorner}}{\text{CompHorner}}$
Measured #cycles ratio	1	2.7 – 3.2	8.5 – 9.7	2.9 – 3.3
Simulated #cycles ratio for superscal.rate = 2	1	3.09	8.49	≈ 2.75
Simulated Optimal IPC	2.5	12.44	1.82	6.83
Optimal IPC (by hand)	1	11	1.6	≈ 6.8

IPC explains the measured performances compared to Flop count

Simulated optimal IPC $\approx$ Optimal IPC by hand for CompHorner and DDHorner

- Small differences comes from instructions *vs.* flop
- Consequence: the number of cycles for CompHorner equals the Horner one for superscalar rate larger than 16

With a binary instrumentation tool: PinTool

Simulating the ideal processor with Pin (Intel)

`http://www.pintool.org`

Pin allows us to instrument the binary code (for Intel architectures)

- Insert extra code into the program to collect run-time information
- Execution trace: the instrumentation applies dynamically

Simulating the ideal processor with Pin (Intel)

`http://www.pintool.org`

Pin allows us to instrument the binary code (for Intel architectures)

- Insert extra code into the program to collect run-time information
- Execution trace: the instrumentation applies dynamically

x86 assembler, instrumentation is in color

```
counter++
mov p, Rp     p= p * x;
counter++
mul x, Rp
counter++
mov a, Ra     p = p + a;
counter++
add Ra, Rp
counter++
```

Simulating the ideal processor with Pin (Intel)

`http://www.pintool.org`

Pin allows us to instrument the binary code (for Intel architectures)

- Insert extra code into the program to collect run-time information
- Execution trace: the instrumentation applies dynamically

A PinTool to compute the Optimal IPC: principles

- For every executed instruction i, Pin returns its characteristics: operation code, source register, destination, memory address...
- Resolution of the data dependencies between i and its sources: recovering of cycle $c = \max\{\text{cycle that produces a source for instruction } i\}$.
- Tag i as being executed and its destination as being available at cycle $c+1$,
- increment the instruction counter

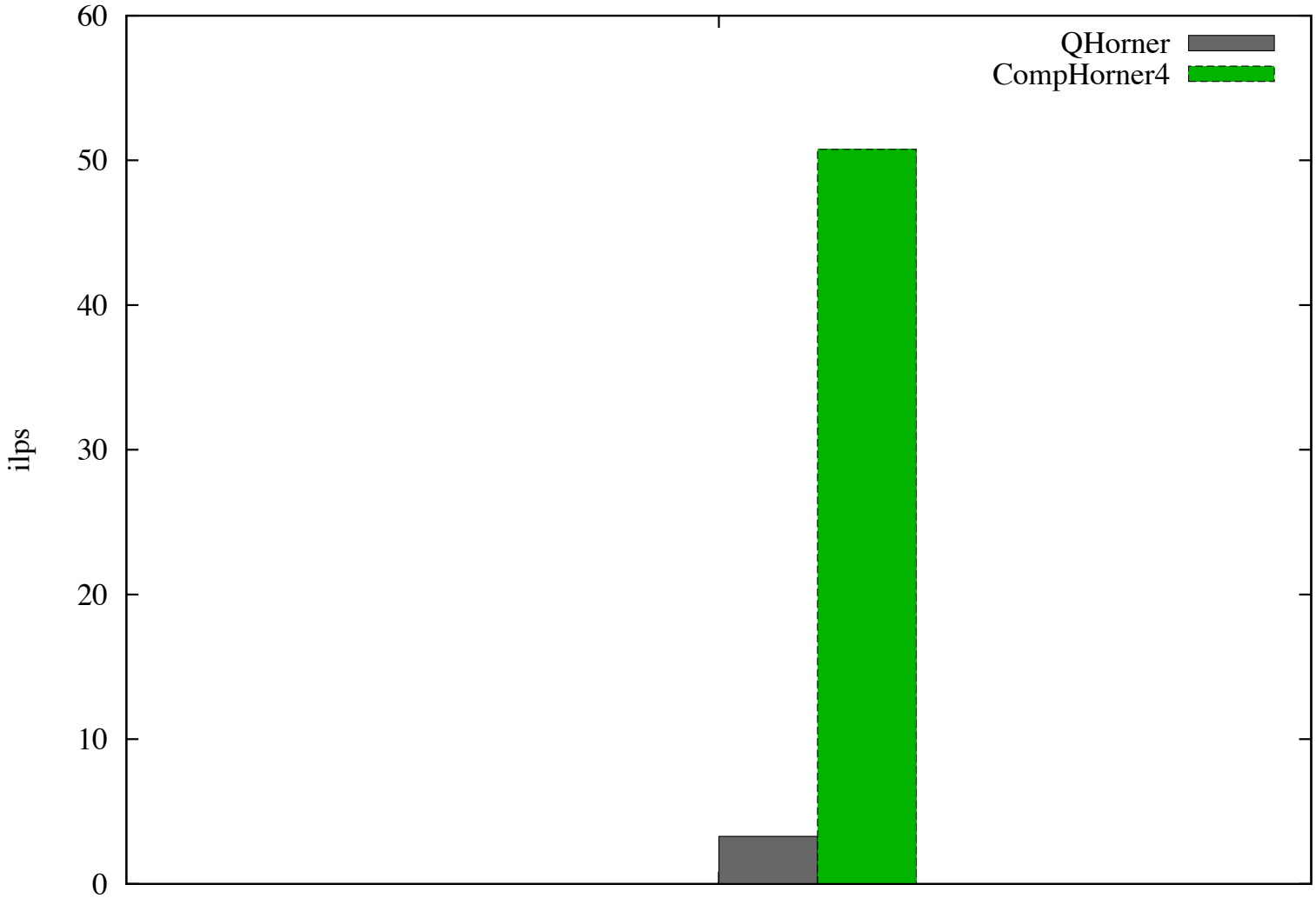
Optimal IPC = # instructions / # cycles

Simulation produces reproducible results

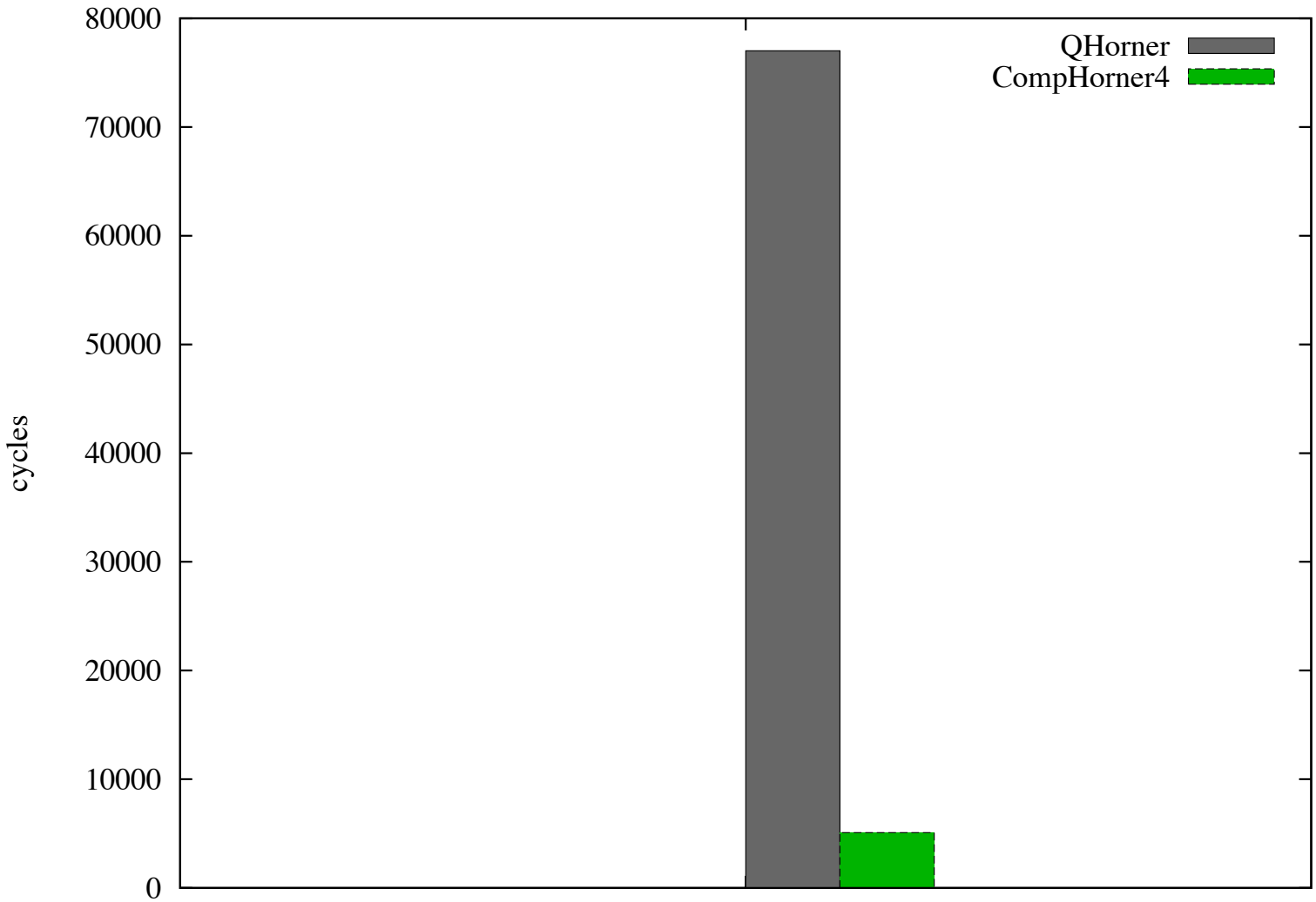
```
start : _start
     start : .plt
          start : __libc_csu_init
               start : _init
                    start : call_gmon_start
                    stop : call_gmon_start::I[13]::C[9]::ILP[1.44444]
                    start : frame_dummy
                    stop : frame_dummy::I[7]::C[3]::ILP[2.33333]
                    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]
```

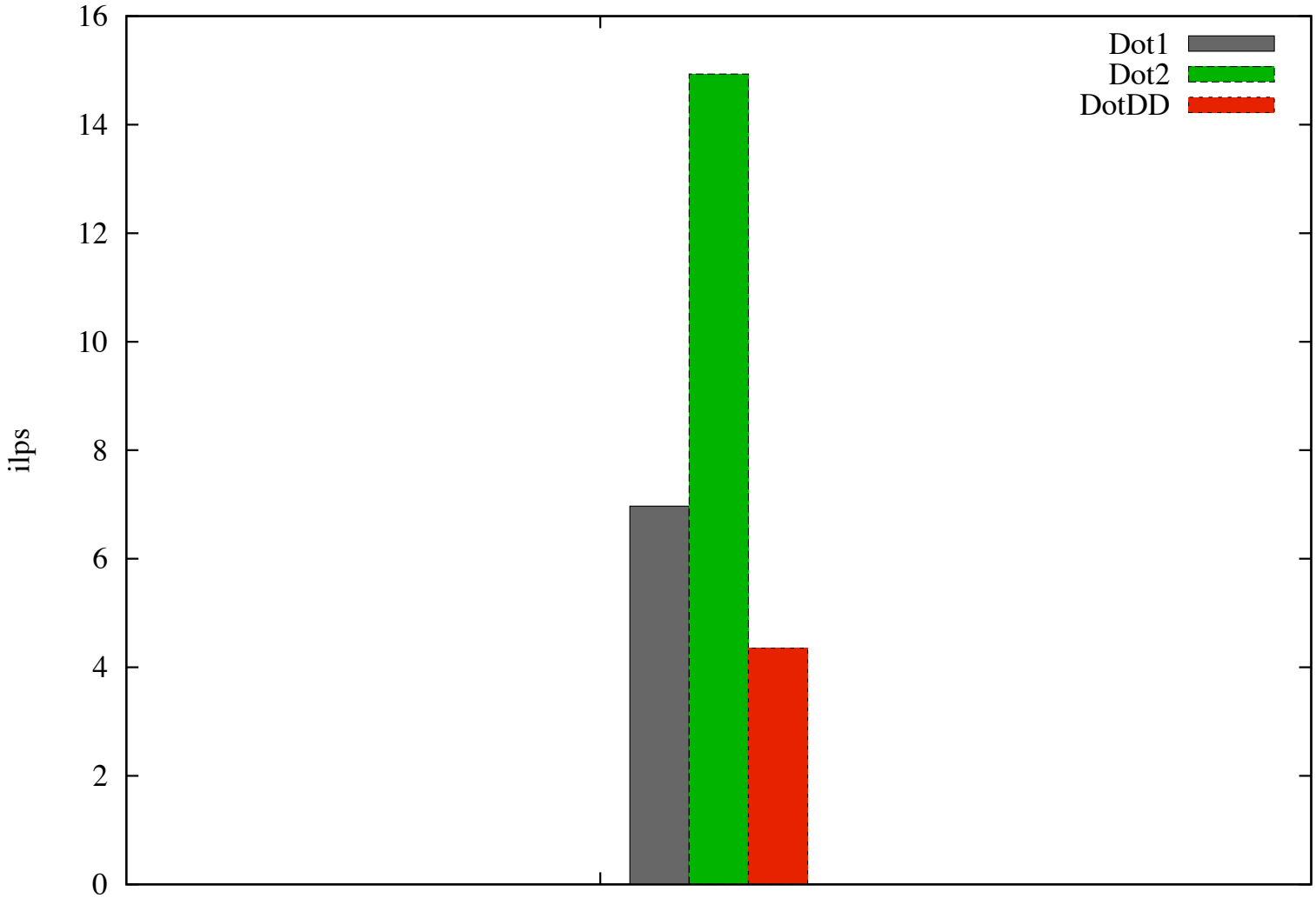
The simulated ideal processor runs CompHorner4



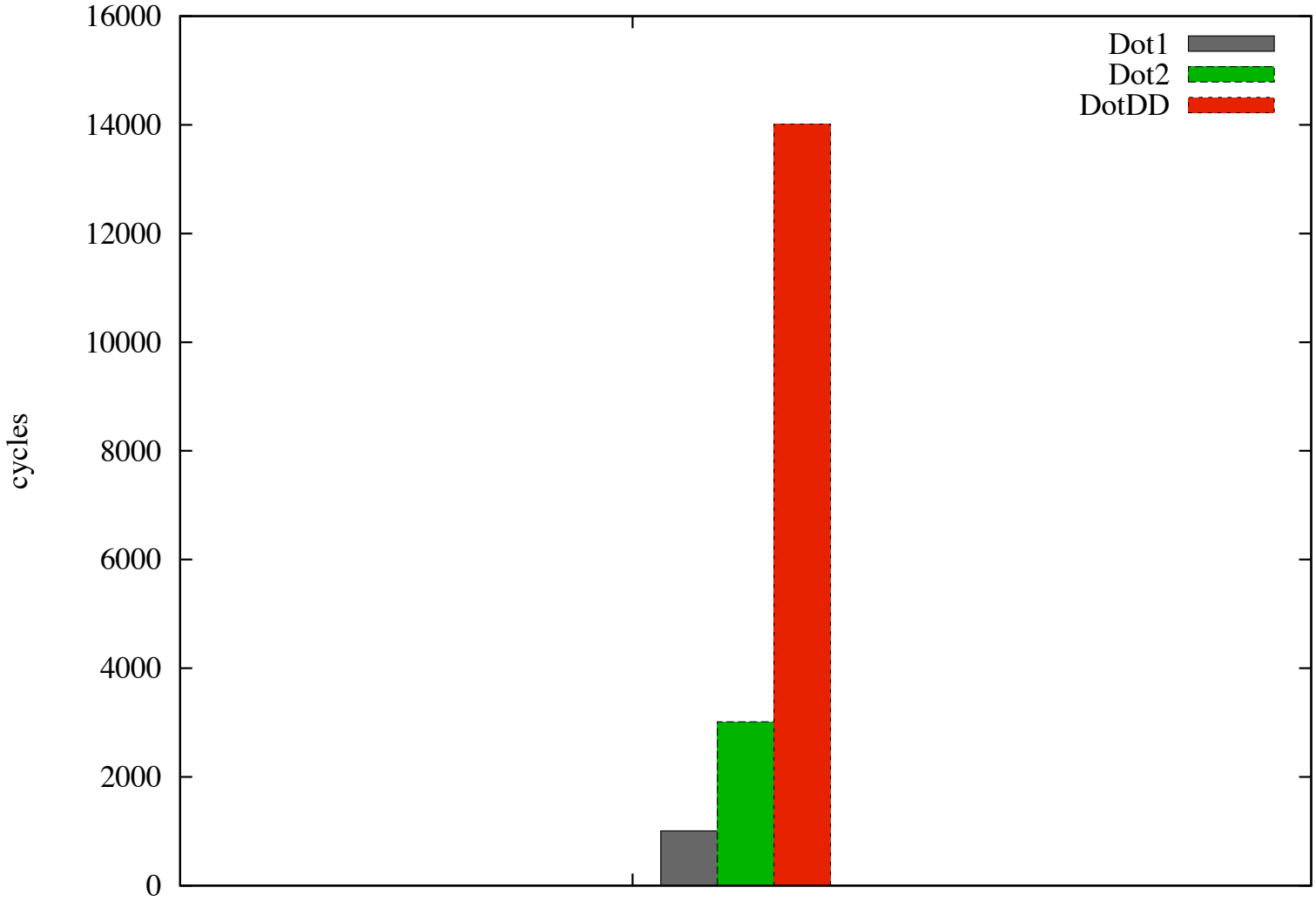
The simulated ideal processor runs CompHorner4



The simulated ideal processor runs accurate dot product (OgRO:05)



The simulated ideal processor runs accurate dot product (OgRO:05)



The simulated ideal processor runs accurate dot product

	Dot	Dot2	DotXBLAS
Measured[2] # cycles ratio Avg. for Pentium4-32	1	5	9.5
Simulated #cycles ratio for superscal.rate = 2 for superscal.rate = 4	1 1	≈ 7 ≈ 3	≈ 9 ≈ 9
Simulated min #cycles ratio	1	≈ 2	≈ 9
Simulated max IPC	4	15	2

- Compensated Dot2 has an impressive high IPC: ≈ 15;
- it will provide twice more accurate dot product for twice more computing times than the original result

[2] From OgRO:05

Outline

Conclusions about simulation

Pros

- Reliable: results are reproducible both in time and location
- Realistic: simulation results can be correlated with measured ones
- Exploratory tool: gives us the taste of the behavior of our algorithms within "tomorrow" processors
- Optimisation tool: simulation provides a simple way to analyse the effect of some hardware constraints
- Useful: providing a very detailed picture of the behavior of the algorithm

Cons . . . at the current state

- It remains some architecture dependencies: the instruction set
- Assembler program or High level programming language?
  IPC and Floating Operations per Cycle may be different for too arithmetically simple algorithms

Todo list

- Focus on Floating Operations per Cycle
- Improve the simulators running-time performance
- Make them available on-line and usable as black-box

Towards computed-assisted proof for performance evaluation

- Pro: simulation provides automatic results
- Cons: reliability of the simulation result depends on the reliability of the compiler and on the simulator

Bibliography and links

John L. Hennessy and David A. Patterson.

Computer Architecture – A Quantitative Approach.

Morgan Kaufmann, 2nd edition, 2003.

Philippe Langlois and Nicolas Louvet.

More instruction level parallelism explains the actual efficiency of compensated algorithms.

Technical Report hal-00165020, DALI Research Team, HAL-CCSD, July 2007.

Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi.

Accurate sum and dot product.

SIAM J. Sci. Comput., 26(6):1955–1988, 2005.

Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi.

Accurate floating-point summation –part I: Faithful rounding.

SIAM J. Sci. Comput., 2008.