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

Bernard Goossens, Philippe Langlois, David Parello, Eric Petit

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

Bernard Goossens, Philippe Langlois, David Parello, and Eric Petit*

*DALI Research Team, University of Perpignan Via Domitia, France* †

**Abstract** We measure and analyze the instruction level parallelism which conditions the running-time performance of core numerical subroutines. We propose PerPI, a programmer oriented tool to fill the gap between high level algorithm analysis and machine dependent profiling tools and which provides reproducible results.

**Keywords** Running-time performance, instruction level parallelism, ideal processor, BLAS, polynomial evaluation, mixed precision

## 1 Introduction

We introduce PerPI, a programmer oriented tool to focus the instruction level parallelism of numerical algorithms. This tool is motivated by results like those presented in Table 1 where two algorithms are compared with respect to flop counts and running-time measures. The first two lines are significant of the algorithm complexity while the last one presents the range of running-times measured for several desktop computers. Such measures are very classical when publishing new core numerical algorithms, *e.g.*, dot product, polynomial evaluation — see entries in [6] for instance. These two algorithms solve the same problem:

| **Measure** | 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 |

**Table 1:** Flop counts and running-times are not proportional

how to double the accuracy of a core numerical subroutine? Such a need appears for example in numerical linear algebra where the accurate implementation of iterative refinement relies on a dot product performed with twice the current computing precision [2]. In Table 1, Eval is the classical Horner algorithm for polynomial evaluation and AccEval1 and AccEval2 are two challenging twice more accurate evaluations (the polynomial degree is $n$). AccEval1 appears to run about three times faster than AccEval2 whereas their flop counts are similar. Such a speedup is interesting for basic numerical subroutines that are used at any parallelism level, and so has to be justified.

Of course only counting the number of flops within an algorithm is not significant of the actual performance of its implementation. This latter depends a lot on other factors as, *e.g.*, parallelism and memory access. Moreover measuring actual running-times is a task that is hard to reproduce and that yields results with a very short life-time since computing environments change quickly. This process is very sensitive to numerous implementation parameters as architecture characteristics, OS versions, compilers and options, programming language, . . . Even when using the same input data set in the same execution environment, measured results suffer from numerous uncertainties: spoiling events (*e.g.*, OS process scheduling, interrupts), non deterministic execution and timings accuracy.

> Measuring the computing time of summation algorithms in a high-level language on today's architectures is more of a hazard than scientific research [6].

This recent quotation seems to us significant of (a call for) a change of practice in the numerical algorithm community. Indeed uncertainty increases as the computer system complexity does, *e.g.*, multicore or hybrid architectures. Even in the community of program and compiling optimization, it is not always easy to trust this experimental process.

> 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? [7]

A last problem comes from the gap between the algorithm design step and the profiling one. The algorithmic step benefits from the abstraction of high level programming languages and, more and more, from the interactivity of integrated developing frameworks such as Matlab. Running-time performance analysis is processed later and in a technically more complex and changing-prone environment. The programmer suffers from the lack of performance indicators, and associated tools, being independent

*Email: `first_name.name@univ-perp.fr`

†This work is partly funded by the ANR Project EVA-Flo ANR-BLAN 06-2-135670 2006.

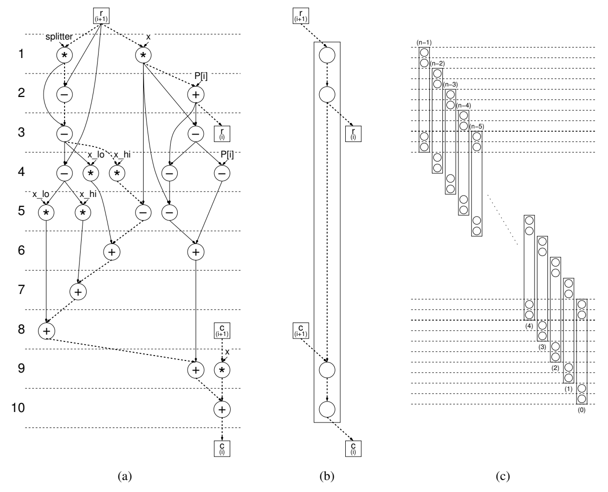
**Figure 1:** Data-flow graph of AccEval1

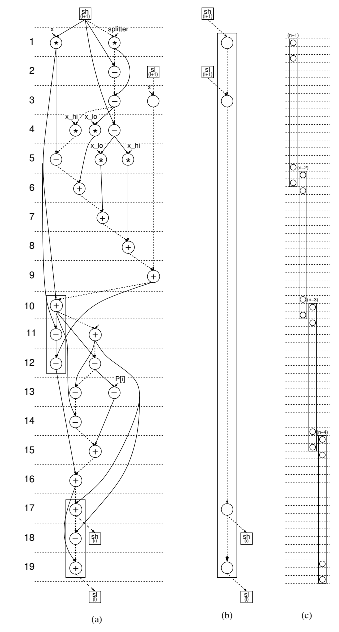
**Figure 2:** Data-flow graph of AccEval2

of the targeted computing architecture that would help him at the algorithmic level to chose more efficient and perennial solutions.

## 2 Analysis principles

In this Section, we describe the principles of our analysis and illustrate it with an introductory pen-and-paper analysis.

### 2.1 Principles

We propose to analyze the instruction level parallelism (ILP) of a program through a simulation of its run with a Hennessy-Patterson ideal machine [1]. The ILP represents the potential of the instructions of a program that can be executed simultaneously. Every current processor exploits program ILP thanks to well known techniques such as pipelining, superscalar execution, prediction, out-of-order execution, dynamic branch prediction or address speculation,... The ideal machine removes all artificial constraints on the ILP. A simulated ideal machine runs the program in a way such that every instruction is scheduled immediately after the execution of the last predecessor on which it depends.

The following example illustrates how to quantify this ILP and what kind of information is useful to understand and improve the potential performance of an algorithm.

### 2.2 A first pen-and-paper analysis

Algorithms presented in Table 1 consist of one loop of length *n*. Figures 1 and 2 represent the data-flow graphs of the two accurate algorithms: (a) represents one iteration, (b) how one iteration depends on the previous one, and (c) the shape of the whole loop (or part of it) [3]. Two consecutive horizontal layers represent two consecutive execution cycles within the ideal machine.

As it was manually performed, the data dependencies analysis has been here restricted to the floating-point operations, *i.e.*, to the algorithmic level description. The whole program instructions will be covered with the PerPI tool further introduced. From these graphs, we count the number of floating-point operations and the number of cycles to run the code, *i.e.*, the total number of nodes and the depth of the (c) graph. The ratio of these values measures the (floating-point) ILP, *i.e.*, the average width of the data-flow graph. These values are reported in Table 2. AccEval1

| **Measure** | Eval | AccEval1 | AccEval2 |
|---|---|---|---|
| **FP ILP** | 1 | $\approx 11$ | $\approx 1.65$ |

**Table 2:** Floating-point ILP as in Table 1

benefits from about 6.66 times more ILP than AccEval2. This certainly justifies that AccEval1 runs faster than AccEval2 on modern processors which are designed to capture some of the ILP. Of course no quantitative correlation with the measured cycles ratios can be done as current processors have limited resources that the ideal machine has not. Nevertheless comparing the FP ILP to the FP count ratios, we deduce that the accurate evaluation AccEval1 would run as fast as the classical Eval on a processor which would capture the whole ILP of this algorithm. The analysis of the graphs also exhibits the origin of such ILP differences. The two algorithms use almost the same groups of operations but AccEval2 suffers from two bottle-necks identified as vertical rectangles on the (a) graph. In this scope, this property will be useful to design other accurate algorithms based on AccEval1 rather than on AccEval2.

## 3 The PerPI tool

We now present the PerPI tool which automatizes this analysis. PerPI currently includes the following facilities: ILP computation, ILP histogram and data flow graph display.

## 3.1 Computing the ILP

The measuring part of PerPI is a Pin tool[5]. It computes $ILP = I/C$, where $I$ is the number of machine instructions run and $C$ is the number of *steps* needed to complete the run. The higher *ILP*, the more parallel the piece of code.

A *step* is defined as the following sequence of operations: for every *runnable* instruction, its source registers are read, its memory read references are loaded, its operations are computed, its destination registers are written and eventually its memory write references are stored.

For example, `addl %eax,4(%ebp)` reads registers $EAX$ and $EBP$, computes $a = EBP + 4$, loads memory referenced by $a$ (assume value $v$ is loaded), computes $r = EAX + v$ and stores $r$ to memory referenced by $a$ (the `addl` instruction could be the translation of a C source code instruction such as `x=x+y` where $x$ is in the function frame on the stack at address $a$ and $y$ is in register EAX).

A step is performed in many cycles in a real machine. However in our tool, a step is considered as atomic to match the ideal machine. As in the example, ILP is the average number of machine instructions run per step. This definition of the ILP removes any micro-architectural details such as latency and throughput. We assume the piece of code is run on the best possible processor, with infinite resources and single cycle latency operators (including memory access and conditional and indirect branch resolution).

An instruction is *runnable* when all the source registers and all the memory read references are ready, *i.e.*, have been written by preceding instructions.

The Pin tool computes ILP as follows. For each instruction of the run, apply the following procedure.

1. For each source register, get the step at which it is updated
2. For each memory read reference, get the step at which it is updated
3. Let $R$ be the latest of all the source register update steps
4. Let $M$ be the latest of all the memory read reference update steps
5. The instruction is run at step $s = max(R,M) + 1$
6. For each destination register, mark it as being updated at step $s$
7. For each memory write reference, mark it as being updated at step $s$

While we compute for each instruction its step $s$, we update the number of run cycles $C$ the following way: $C = max(s,C)$.

For any piece of code, the set of registers and memory references are assumed to be updated at step 0 when the run starts. For reproducibility reasons, the system calls involved in the measured piece of code are not considered.

## 3.2 Analyzing the ILP

The observation part of PerPI consists in histograms and graphs displaying functions. These functions allow the user to zoom in and out of the trace. As before, the graph represents the instructions dependencies where an instruction $j$ depends on an instruction $i$ iff $j$ has a source provided by $i$ ($j$ reads a register or a memory word $x$ written by $i$ and no instruction between $i$ and $j$ writes to $x$). The histogram represents the variation of the ILP along the steps.

The histogram tool is useful to locate the good (high ILP) and the bad (low ILP) portions of the code run. The graph tool is useful to analyze why a code has a high or low ILP as the example illustrates.

## 3.3 Results examples

We present PerPI results for some accurate summation algorithms introduced in [4] and previous polynomial evaluation algorithms. Sum2 and SumXBLAS are respectively similar to AccEval1 and AccEval2. These algorithms are implemented as C functions and are called in a main part. From a practical point of view, binary files are submitted to PerPI through a graphical interface and then some menu items generate the following outputs.

We first illustrate the ILP measured with Figure 3. Every called subroutine is analyzed, *i.e.*, PerPI returns the number of machine instructions $I$, the number of step $C$ and the corresponding *ILP*. The given results are those obtained after a single run as, as we have mentioned earlier, the computed values are fully reproducible.

```
start:   _start
...
start:    main
 start:    init
 stop:     init   ::I[534]::C[105]::ILP[5.08]
 start:    Sum
 stop:     Sum  ::I[511]::C[105]::ILP[4.86]
 start:    Sum2
 stop:     Sum2   ::I[1617]::C[214]::ILP[7.55]
 start:    SumXBLAS
 stop:     SumXBLAS   ::I[2097]::C[898]::ILP[2.33]

stop:     main   ::I[4812]::C[1226]::ILP[3.92]
...
Global ILP  ::I[4919]::C[1279]::ILP[3.84]
```

**Figure 3:** A call graph with the ILP of three summation algorithms for 100 summands

The matching histograms are presented in Figures 4 and 5 – legends are not displayed here. Zooms are available, *e.g.*, Figure 6. In this case the red bars indicate floating point operations while purple ones are data transfers. These histograms exhibit the regularity of the two algorithms ILP and Sum2 better efficiency.

The last presented outputs are the data-flow graphs Figures 7 and 8. As for the introductory ones, cycles are on the Y-axis. When zooming in some interesting parts the corresponding program instructions are displayed to help the programmer analyzing his code.

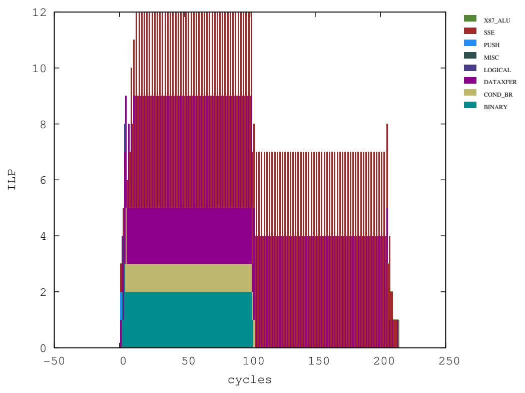

**Figure 4:** Sum2 histogram for 100 summands

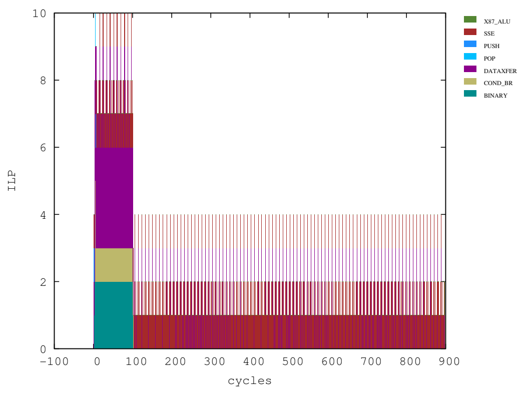

**Figure 5:** SumXBLAS histogram for 100 summands

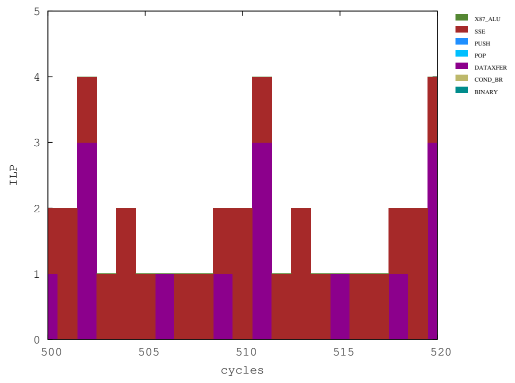

**Figure 6:** Zoom of Figure 4

## 4 Conclusions and current work

The presented performance analysis and its PerPI tool aim to fill the gap between the high level algorithm analysis and machine dependent profiling tools. We show from examples on some core numerical algorithms that our preliminary results are interesting enough and we estimate they validate the proposed approach. We highlight the fact that these results are reproducible. They should help the programmer to both justify his measured performances and improve his algorithm. The presented version of PerPI will be publicly available soon. Work is in progress to extend the analysis facilities implemented in PerPI, as for example

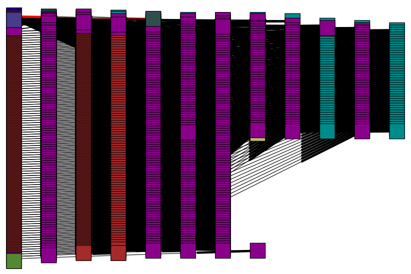

**Figure 7:** Sum2 data flow graph for 100 summands

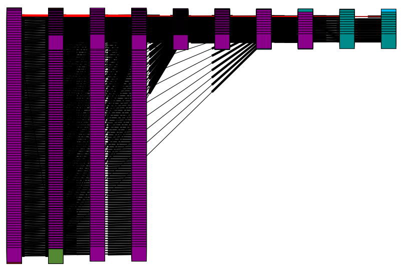

**Figure 8:** SumXBLAS data flow graph for 100 summands

identifying and observing the longest dependency chains of instructions or constraining the ideal machine. As PerPI is based on Pin it concerns x86 machine code only. We are investigating in what extent the machine language impacts the ILP measure. This is out of the scope of this paper — it is easy to find examples in which a CISC x86 piece of code has a higher ILP than its equivalent RISC MIPS or PowerPC and conversely.

## References

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 2003.

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

[3] P. Langlois and N. Louvet. More instruction level parallelism explains the actual efficiency of compensated algorithms. Technical report, DALI Research Team, 2007. http://hal.archives-ouvertes.fr/hal-00165020.

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

[5] Pin. URL = `http:www.pintool.org`.

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

[7] S. Touati. Towards a Statistical Methodology to Evaluate Program Speedups and their Optimisation Techniques. Technical report, PRISM, UVSQ, 2009. http://hal.archives-ouvertes.fr/hal-00356529/en/.