PerPI: A Tool to Measure Instruction Level Parallelism

Bernard Goossens, Philippe Langlois, David Parello, Eric Petit

PerPI: A Tool to Measure Instruction Level Parallelism

Bernard Goossens, Philippe Langlois, David Parello and Eric Petit

DALI Research Team, University of Perpignan Via Domitia,
52 av. P. Alduy, F-66860 Perpignan cedex, France.
first_name.name@univ-perp.fr
http://webdali.univ-perp.fr

Abstract. We introduce and describe PerPI, a software tool analyzing the instruction level parallelism (ILP) of a program. ILP measures the best potential of a program to run in parallel on an ideal machine – a machine with infinite resources. PerPI is a programmer-oriented tool the function of which is to improve the understanding of how the algorithm and the (micro-) architecture will interact. PerPI fills the gap between the manual analysis of an abstract algorithm and implementation-dependent profiling tools. The current version provides reproducible measures of the average number of instructions per cycle executed on an ideal machine, histograms of these instructions and associated data-flow graphs for any x86 binary file. We illustrate how these measures explain the actual performance of core numerical subroutines when measured run times cannot be correlated with the classical flop count analysis.

Keywords: Run time performance, instruction level parallelism, ideal processor, BLAS, polynomial evaluation, mixed precision

1 Introduction

1.1 Motivation

We introduce PerPI, a programmer-oriented tool focusing the instruction level parallelism of numerical algorithms. This tool is motivated by results like those

Measure	Eval	AccurateEval1	AccurateEval2
Flop count	2n	$22n + 5$	$28n + 4$
Flop count ratio (/Eval)	1	≈ 11	≈ 14
Measured #cycles ratio (/Eval)	1	2.8 – 3.2	8.7 – 9.7

Table 1. Flop counts and run times are not proportional

presented in Table 1 where two algorithms, AccurateEval1 and AccurateEval2, are respectively compared to a third one, Eval [3]. The three algorithms evaluate a polynomial of degree n. Eval is the classical Horner algorithm, AccurateEval1 and AccurateEval2 are two competing evaluations which are both twice more

accurate. These two algorithms solve the same problem: how can we double the accuracy of a core numerical subroutine? Such a need appears also, for example, in numerical linear algebra where an accurate iterative refinement relies on a dot product performed with twice the current computing precision [2].

Table 1 presents the flop counts and the ratios of flop counts and run times for the two accurate algorithms over Eval ones. The first two lines are significant for the algorithm's complexity, while the last one presents the range of run times measured on several desktop computers. Such measures are quite familiar when publishing new core numerical algorithms, *e.g.*, floating-point summation, dot product, polynomial evaluation — see entries in [8] for instance. Here AccurateEval1 appears to run about three times faster than AccurateEval2, whereas their flop counts are similar. Such a speedup is important for basic numerical subroutines that are used at any parallelism level. This gap between the classical manual analysis of the abstract algorithms (flop counts) and the measures provided by automatic profiling tools (cycle counts, with [6] for instance) has to be justified.

Of course, merely counting the number of flop within an algorithm does not fully explain the actual performance of its implementation, which depends on other factors such as parallelism and memory access. Moreover, measuring actual run times is hard to reproduce and yields results with a very short life-time since computing environments evolve fast. This process is very sensitive to numerous implementation parameters such as architectural and microarchitectural characteristics, OS versions, compilers and options, programming language, etc. Even if the same data test is used in the same execution environment, measured results suffer from numerous uncertainties: spoiling events (*e.g.*, OS process scheduling, interrupts), non-deterministic execution and accuracy of the timings [11].

> Measuring the computing time of summation algorithms in a high-level language on today's architectures is more of a hazard than scientific research [8].

We believe that this recent quotation is significant for (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 for four decades, why don't we observe execution times approaching to zero? [10]

A last difficulty 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. Run time performance analysis is a later step process, and it takes place in a technically more complex and change-prone

environment. The programmer suffers from the lack of performance indicators, and associated tools, being independent of the targeted computing architecture that would help at the algorithmic level to choose the most efficient and long-lasting solutions.

2 Analysis: principles and pen-and-paper example

2.1 Principles

We propose to analyze the instruction level parallelism (ILP) of a program by simulating its run with a Hennessy-Patterson ideal machine [1]. An ideal machine has infinite resources: renaming registers, perfect branch predictor, perfect memory disambiguation. As a result, running a code on an ideal machine is like having at hand the full trace and picking up from this trace instructions as soon as their sources are available. In such a way, the run is ordered according to the only producer/consumer dependencies.

ILP represents the best potential of the instructions of a program that can be executed simultaneously. Every current processor exploits program's ILP thanks to well-known techniques such as pipelining, superscalar execution, out-of-order execution, dynamic branch prediction or address speculation, etc. The ideal machine removes all artificial constraints on ILP (registers, memory, control flow), so it runs the program in such a way that every instruction is scheduled immediately after the execution of the latest producer 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

The algorithms presented in Table 1 consist of one loop n times iterated. Figure 1 represents the data-flow graphs of the two accurate algorithms: (a) one iteration, (b) one iteration depending on its predecessor, and (c) the shape of the n iterations (or part of it) [3]. In each of the three parts of the figure, two consecutive horizontal layers represent two consecutive execution cycles within the ideal machine. To be performed manually, the data dependency analysis has been restricted to the floating-point operations, *i.e.*, to the algorithmic level description.

Measure	Eval	AccurateEval1	AccurateEval2
FP ILP	1	≈ 11	≈ 1.65

Table 2. Floating-point ILP as in Table 1

From these graphs, we count the number of floating-point operations and the number of cycles to run them, *i.e.*, the total number of nodes and the depth of the (c) graph. The ratio of these values measures the (floating-point) ILP,

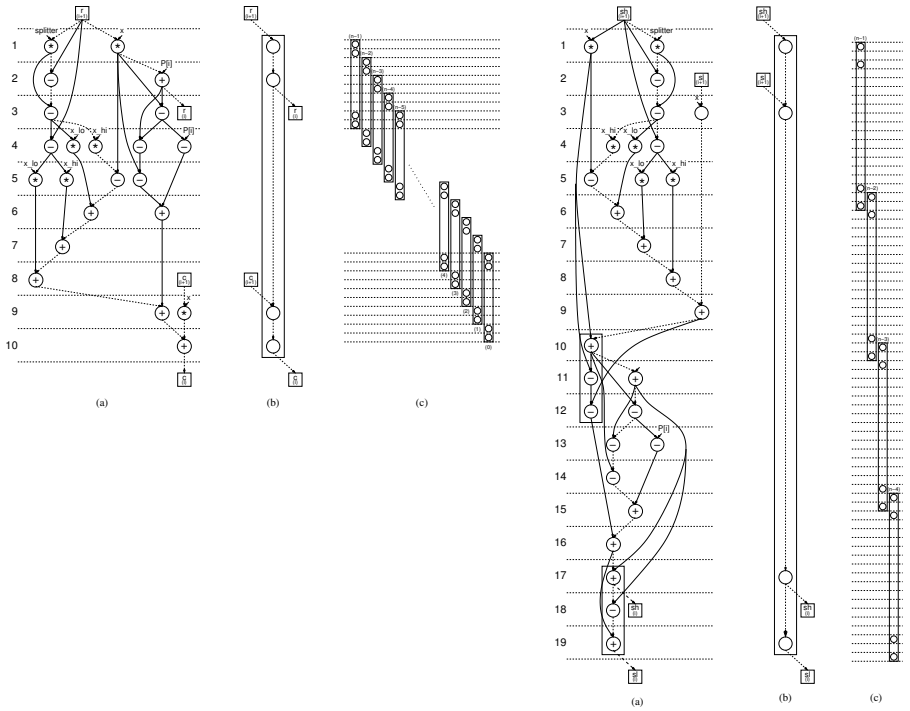

Fig. 1. Data-flow graphs of the main loop in AccurateEval1 (left) and AccurateEval2 (right). (a) one iteration, (b) its dependencies and (c) n iterations (part of it for the right one)

i.e., the average width of the full loop data-flow graph. These values are reported in Table 2. AccurateEval1 benefits from about 6.66 times more ILP than AccurateEval2. This certainly justifies that AccurateEval1 runs faster than AccurateEval2 on modern processors that are designed for exploiting this ILP. Of course no quantitative correlation with the measured cycles ratios can be done: current processors have limited resources compared to the ideal machine, and this pen-and-paper analysis only considers floating-point operations. In the next Section we present the PerPI tool which builds the full data dependency graph, including all the instructions in the trace, being floating point computations or integer control. Nevertheless, comparing this floating point ILP (Table 2) and the floating point count ratios (Table 1) of this manual analysis, we deduce that the accurate evaluation AccurateEval1 will run as fast as Eval on a processor that will exploit the whole ILP of this algorithm.

The graph analysis also exhibits the origin of such ILP differences. The two algorithms use almost the same groups of operations, but AccurateEval2 suffers from two bottle-necks identified as vertical rectangles on the (a) graph. A detailed analysis is presented in [3]. In this perspective, this property will be useful to design other accurate algorithms more inspired by AccurateEval1 than by AccurateEval2.

3 The PerPI tool

We now present the PerPI tool that automates this ILP analysis. PerPI currently includes the following facilities: ILP computation, ILP histogram and data flow graph displays.

3.1 Computing ILP

The measuring part of PerPI is a Pin tool [7]. Pin [4] is an Intel (R) free programmable tool. Pin consists of an engine which instruments any code at run time with user-defined measurement routines. PerPI is a set of routines aiming at computing the run code ILP. The ILP is computed while the real code is run. The examining routine gives the control of the examined code for a single instruction run and recovers control to update its examination statistics. This back and forth execution is continued until the examined code has been fully scanned. At each step of the examination, PerPI computes the run cycle of the examined instruction, increments the number of instructions run so far and possibly updates the highest run cycle. In such a way, PerPI 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 the 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 in turn, apply the following procedure (*i.e.*, the procedure is applied to the full trace, in order).

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 $c = max(R, M) + 1$
6. For each destination register, mark it as being updated at step c
7. For each memory write reference, mark it as being updated at step c

While we compute the steps c, we adjust the step that has been last computed, giving the number of cycles of the run $C = \max(c)$.

For reproducibility, the system calls involved in the measured piece of code are not considered.

Table 3 illustrates how this algorithm computes the ILP of the expression `(a+b)+(c+d)`. The first instruction (eax=a, line 2) has its source ebp available at cycle 0 (line 1, column 7). It is run and it updates its destination register eax, making it available for later instructions at cycle 1 (line 2, column 3). Column 8 is the instruction number (I). Column 9 is its run cycle (c) and column 10 is the greatest run cycle (C). The last instruction (edx+=ebx) reads its sources ebx and edx at cycle 2 (look at the preceding line) and so is run and updates edx at cycle 3 (last line, column 6). There are 6 instructions ($I = 6$, last line column 8) run in 3 cycles ($C = 3$, last line column 10) on an ideal machine, which gives an ILP of $6/3 = 2$. The ideal machine runs this fragment of code at an average rate of two instructions per cycle.

		Availability as source register							
Instruction	Semantic	eax	ebx	ecx	edx	ebp	I	c	C
		0	0	0	0	0	0	0	0
`mov eax,DWP[ebp-16]`	eax=a	1	0	0	0	0	1	1	1
`mov edx,DWP[ebp-20]`	edx=b	1	0	0	1	0	2	1	1
`add edx,eax`	edx+=eax	1	0	0	2	0	3	2	2
`mov ebx,DWP[ebp-8]`	ebx=c	1	1	0	2	0	4	1	2
`add ebx,DWP[ebp-12]`	ebx+=d	1	2	0	2	0	5	2	2
`add edx,ebx`	edx+=ebx	1	2	0	3	0	6	3	3

Table 3. ILP computation yields ILP $= I/C = 2$, when evaluating $(a + b) + (c + d)$.

3.2 Analyzing facilities

The analysis part of the tool consists in histogram and graph displaying functions. These functions allow the user to zoom in and out of the trace. As in the example, the graph represents the instruction 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 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 illustrated in Section 4.

3.3 How to interpret the measured ILP?

ILP is defined as the number of machine instructions run divided by the number of ideal machine cycles needed to run them all on the ideal machine.

This definition shows that ILP is architecture-dependent. The number of instructions depends both on the machine language employed and the compiler.

For example, it is easy to exhibit examples where a RISC-like (*e.g.*, PowerPC, ARM, MIPS, SPARC) translation of a high-level code sequence shows a higher ILP than its CISC-like (*e.g.*, x86) equivalent. Table 4 illustrates this for the C code instruction `a += b;`. The *c* column depicts the cycle during which the instruction is run (starting from cycle 1).

c	RISC		CISC	
	Instruction	Semantic	Instruct.	Semantic
1	`load a,ra`	load memory a into register ra	`mov b,rb`	load mem. b into reg. rb
1	`load b,rb`	load memory b into register rb		
2	`add ra,rb,ra`	ra:=ra+rb	`add rb,a`	add reg. rb to mem. a
3	`store ra,a`	store register ra into memory a		

Table 4. ILP of `a+=b;` equals 4/3 for the RISC translation and 1 for the CISC one.

In this example, we have ILP(RISC) > ILP(CISC). This comes from the load/store model inherent to the RISC-like machine languages in which an instruction is either a memory access (load or store) or a computation involving registers only. In a CISC-like machine language, an instruction may involve both a memory access and a computation. This difference results in a RISC translation having more instructions than its CISC equivalent, leading to a possibly higher ILP (if more instructions are run in the same number of cycles).

However, we may notice that #cycles(CISC) < #cycles(RISC), meaning that the CISC code can be run faster than the RISC one. We may also notice that #instructions(CISC) < #instructions(RISC), meaning that the CISC code needs less resources than the RISC one.

Another difference between RISC and CISC leads to the opposite ILP ranking. Any x86 machine language computing instruction has an accumulating destination whereas in any RISC machine language, the destination may be distinct from the sources. As a consequence, a succession of computations may be translated in less instructions in a RISC language than in a CISC language.

A second example illustrating the preceding remark is given with the translation of $x = |a - b|$ from the C code sequence `x=(a-b>=0)?(a-b):(b-a);`. Corresponding RISC and CISC language translations are presented in Table 5. In this second example, we have ILP(RISC) < ILP(CISC). We also have #cycles(RISC) < #cycles(CISC) and #instructions(RISC) < #instructions(CISC).

These two examples show that ILP should not be taken as the ultimate code quality factor. ILP is dependent on the architecture style (RISC vs CISC, 2-operands vs 3-operands instructions). A high ILP is not synonymous with a fast run but rather with a run which can fill the processor parallel units.

c	RISC		CISC	
	Instruction	Semantic	Instruct.	Semantic
1	`load a,ra`	load mem. a into reg. ra	`mov a,ra`	load mem. a into reg. ra
1	`load b,rb`	load mem. b into reg. rb	`mov b,rb`	load mem. b into reg. rb
2	`sub ra,rb,rx`	rx:=ra-rb	`mov rb,rc`	rc:=rb
2			`sub ra,rb`	rb:=ra-rb
3	`csub (rx<0), 0,rx,rx`	rx:=(rx<0)?(0-rx):rx	`sub rc,ra`	ra:=rc-ra
4	`store rx,x`	store reg. rx into mem. x	`cmovge ra,rb`	rb:=(b-a≥0)?ra:rb
5			`mov rb,x`	x:=rb

Table 5. ILP of $x = |a - b|$ equals 1.25 for RISC and 1.4 for CISC.

4 Examples of results

We present PerPI results for some accurate summation algorithms introduced in [5, 9, 8] and the previous polynomial evaluation algorithms. Sum2 and SumXBLAS are in that sense similar to AccurateEval1 and AccurateEval2. 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 measure with Figure 2. Every called subroutine is analyzed, *i.e.*, PerPI returns the number of machine instructions I, the number of steps C, and the corresponding ILP. One run is enough since these values are reproducible.

```
Sum       ::I[511] ::C[105]::ILP[4.86]
Sum2      ::I[1617]::C[214]::ILP[7.55]
SumXBLAS  ::I[2097]::C[898]::ILP[2.33]
```

Fig. 2. ILP measure for the three summation algorithms from [5] (100 summands)

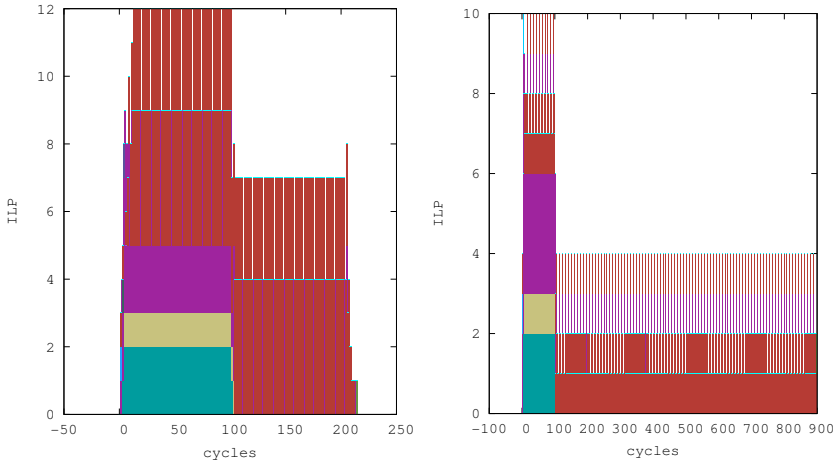

Fig. 3. ILP histograms for Sum2 and SumXBLAS and 100 summands

Corresponding histograms are presented in Figure 3 and can be zoomed in as in Figure 4. This latter exhibits the color significance. In this case the red bars correspond to floating point operations, while purple ones are data transfers. These histograms exhibit the regularity of the ILP of the two algorithms and the better efficiency of Sum2.

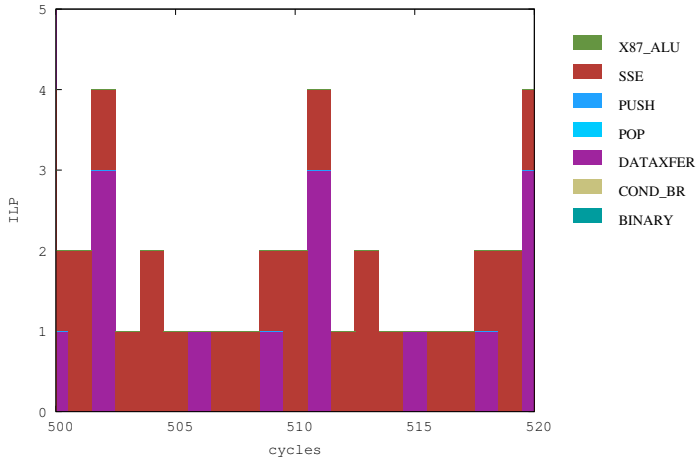

Fig. 4. Zoom of Figure 3 (SumXBLAS) with corresponding instruction type

The shape of the histograms starts with a high ILP part which comes from the control flow instructions. They usually are independent of the data flow computation (they control it) and so can all of them start simultaneously on the ideal machine. Branch, increment and compare (loop control) instructions are located only in this initial part of the histogram. This part serves also as a data flow ILP increasing period as the precomputed control flow instructions launch the data flow instructions which depend on them. After this, we find a data flow plateau (more uniformly colored) and an ILP decreasing end part.

The last outputs are the data-flow graphs presented in Figure 5. This output automates the pen-and-paper results of Figure 1; cycles are on the Y-axis. After zooming into interesting parts of this graph, corresponding program instructions are displayed in such a way that the programmer can analyze the code.

In Figure 6 we display a zoom into the innermost loop for two other accurate summation algorithms introduced in [9, 8], resp. AccSum and FastAccSum. The count of the floating-point operations suggests that FastAccSum should run about 30% faster than AccSum. Performance counter timing of some implementation confirms this speed-up – as in [8] we measure it for instance using gcc -O3 on an Intel Core 2. Nevertheless PerPI yields measures and data flow graphs that exhibit a higher degree of parallelism in one of the innermost loops of AccSum. This parallelism was not automatically detected by the previously mentioned implementation (as the assembly code reveals). A classical way to transform ILP into data parallelism is vectorization, here using SSE instructions. Such an

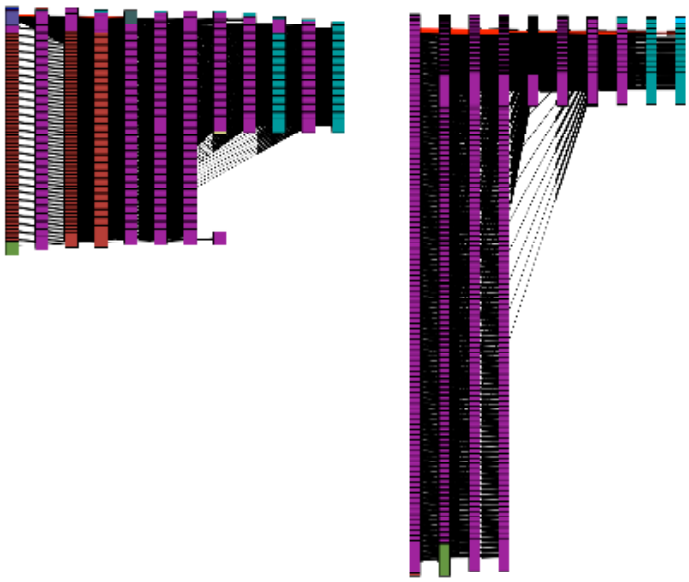

Fig. 5. Sum2 and SumXBLAS data flow graphs for 100 summands (Y-axis: cycles, vertical scales reduce the height of the right one)

improved implementation of AccSum now exhibits an average speed-up of 1.7 compared to FastAccSum[1] – while no cache size limitation applies. The AccSum ILP potential could be caught by vectorization while the lack of ILP in FastAccSum did not give any advantage compared to a vectorized implementation. This analysis also tells us that this algorithm may even benefit from larger vector microarchitectures as the ones present today in GPUs or tomorrow in AVX units in Intel corei7 2011 releases. It still has some ILP left.

5 Conclusions and current work

The presented performance analysis and its PerPI tool aim to fill the gap between high level algorithm analysis and machine-dependent profiling tools. We illustrate on some core numerical algorithms that the first results are interesting and validate the proposed approach. These results are reproducible and help the programmer both to justify the measured performances and to improve the algorithm. As PerPI is based on Pin, it handles x86 machine code only. We have commented on how the machine language has an impact on the ILP measure. 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 identifying longest dependency instruction chains or introducing constraints within the ideal machine.

[1] A speed-up of 1.3 has also been identified later with the newest version of the icc compiler without having to modify the source code. This again illustrates how the measured run times are dependent of the compiler and its versions.

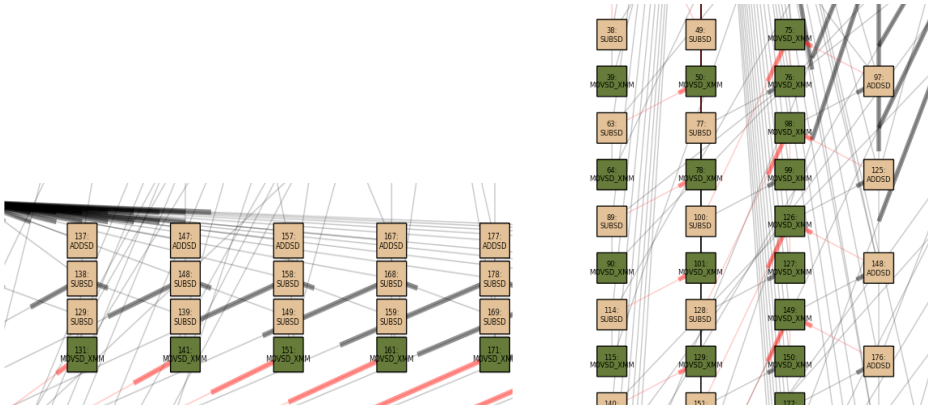

Fig. 6. Zoom into the dataflow graph of corresponding instructions in an innermost loop for two accurate summation algorithms from [9, 8] (Y-axis: cycles). AccSum (left) exhibits more ILP than FastAccSum (right). The added ILP can be exploited thanks to vector units (SSE). The left zoom is actually only partially displayed (it is to be continued on each side with the same 4-steps shape), while the right zoom displays the full (but more limited) width of the loop for FastAccSum.

References

1. J. L. Hennessy and D. A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.
2. N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002.
3. Ph. Langlois and N. Louvet. More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms. Technical Report, DALI Research Team, 2007. URL = `http://hal.archives-ouvertes.fr/hal-00165020`.
4. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, 190–200, 2005.
5. T. Ogita, S. M. Rump, and S. Oishi. Accurate Sum and Dot Product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005.
6. PAPI. URL = `http://icl.cs.utk.edu/papi`.
7. Pin. URL = `http://www.pintool.org`.
8. S. M. Rump. Ultimately fast accurate summation. *SIAM J. Sci. Comput.*, 31(5):3466–3502, 2009.
9. 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.
10. S. Touati. Towards a Statistical Methodology to Evaluate Program Speedups and their Optimisation Techniques. Technical Report, PRISM, UVSQ, 2009. URL = `http://hal.archives-ouvertes.fr/hal-00356529/en/`.
11. V. Weaver and J. Dongarra. Can Hardware Performance Counters Produce Expected, Deterministic Results? 3rd Workshop on Functionality of Hardware Performance Monitoring, Atlanta, GA, December 4, 2010.