

International Parallel Tools
Workshop 2026, October 02, 2026



Friedrich-Alexander-Universität
Erlangen-Nürnberg



WINIC: An Automatic Cycle-Accurate Multi-Platform Microbenchmarking Tool for Instruction Performance Metrics

Tobias Rühl, Jan Laukemann, Georg Hager, Gerhard Wellein

Erlangen National High Performance Computing Center (NHR@FAU)

Outline

- Motivation
 - Usecases for
 - Sources of

Instruction Performance Metrics
- Microbenchmarking
 - Runtime-based measurements
 - Throughput / Latency measurement methodology
- Results
 - Operand-based latencies
 - Comparison to other sources
 - Case Study
 - Limitations / Future Work

Instruction Performance Metrics – Usecase

- Compilers
 - Enable architecture specific optimizations
- Static performance analysis
 - Enable in-core performance engineering tools like OSACA, uiCA, llvm-mca, CQA

[1-3]

Instruction Performance Metrics – Usecase

■ Example: Gauss-Seidel

```
int NI, NK;
double phi [NI][NK];

for (int i=1; i<NI-1; ++i) {
    for (int k=1; k<NK-1; ++k) {
        phi[i][k] = 0.25 * (
            phi[i-1][k]
            +phi[i][k-1]          +phi[i][k+1]
            +phi[i+1][k]
        );
    }
}
```

What is the latency of this chain
on a specific microarchitecture?

<https://godbolt.org/z/anWjG74fh>

```
...
.L5:
    addsd    xmm0, QWORD PTR [rsi+rax*8]
    addsd    xmm0, QWORD PTR [rdx+8+rax*8]
    addsd    xmm0, QWORD PTR [rcx+rax*8]
    mulsd    xmm0, xmm1
    movsd    QWORD PTR [rdx+rax*8], xmm0
    add      rax, 1
    cmp      rax, rdi
    jne      .L5
...
```

GCC 15.2; -O3; inner loop

Instruction Performance Metrics – Sources

- Documentation
 - Not standardised, rarely machine readable
 - Often not publicly available
- Microbenchmarking
 - Manual measurements (ibench, asmbench)
 - Automatic microbenchmarking tools

arm

Arm® Neoverse™ V2 Core

Revision: r0p2

Software Optimization Guide

Non-Confidential

Issue 3.0

Copyright © 2021, 2022 Arm Limited (or its affiliates). PJDOC-466751330-593177

All rights reserved.

3.26 SVE floating-point instructions

Table 3-25 SVE floating-point instructions

Instruction Group	SVE Instruction	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Floating point absolute value/difference	FABD, FABS	2	4	V	-
Floating point arithmetic	FADD, FADDP, FNEG, FSUB, FSUBR	2	4	V	-

[4,6,7]

Instruction Performance Metrics – Sources

		WINIC	nanoBench ^[5]	iBench ^[6]	asmbench ^[7]	llvm-exegesis ^[8]	MAQAO ^[9]
 platform	x86	✓	✓	✓	✓	✓	✓
	AArch64	✓	✗	✓	✓	✓	— §
	RISC-V	✓	✗	✗	✗	✓	✗
 metrics	throughput	✓	✓	✓	✓	✓	✓
	latency	✓	✓	✓	✓	✓	✓
	operand-based latency	✓	✓	✓	✗	✗	✗
	port usage	✗	✓	✗	✗	— *	✓
	user-space	✓	✓	✓	✓	✓	— ‡
 usability	memory instructions	✓	✓	✓	— †	✓	— ‡
	PMU independent	✓	✗	✓	✓	✗	✗

*Supports only x86

†Supports only basic load/store operations

‡Lack of documentation

§Beta

Microbenchmarking – Runtime-based approach

- Generate loop body with instances of the instruction to measure
- Execute with $\sim 10^6$ iterations and measure total runtime
- Calculate cycles / instruction using fixed clock frequency

Microbenchmarking – Throughput

- Repeat instruction, choose different registers for r/w operands

ADD r64 (r/w) , r64 (r)

loop:

add rax, rcx

add rdx, rcx

add rsi, rcx

add r9, rcx

add r10, rcx

...

inc i

cmp i, N

j1 loop

} overhead?

Microbenchmarking – Runtime-based measurements

Loop code causes overhead!

Normal

Unrolled

```
loop:
  add    rax, rcx
  add    rdx, rcx
  add    rsi, rcx
  add    r9, rcx
  add    r10, rcx
  inc    i
  cmp    i, N
  jl     loop
```

body

overhead

```
loop:
  add    rax, rcx
  add    rdx, rcx
  add    rsi, rcx
  add    r9, rcx
  add    r10, rcx
  add    rax, rcx
  add    rdx, rcx
  add    rsi, rcx
  add    r9, rcx
  add    r10, rcx
  inc    i
  cmp    i, N
  jl     loop
```

body

body

overhead

$$T_{Unrolled} - T_{Normal} = T_{body}$$

Microbenchmarking – Throughput

- Repeat instruction, choose different registers for r/w operands

ADC **r64 (r/w)** , **r64 (r)** , **flags (r/w)**

ADD with carry

```
loop:
  adc    rax, rcx #r/w flags
  adc    rdx, rcx #r/w flags
  adc    rsi, rcx #r/w flags
  adc    r9,  rcx #r/w flags
  adc    r10, rcx #r/w flags
  ...
  inc    i
  cmp    i, N
  jl     loop
```

This does not
measure throughput!

Solution: Interleave a
dependency breaking
instruction

Microbenchmarking – Throughput

- Repeat instruction, choose different registers for r/w operands

```
ADC r64(r/w), r64(r), flags(r/w)
TEST r64(r), r64(r), flags(w)
```

```
loop:
  test    rdx, rsi #w flags
  adc     r9, rcx  #r/w flags
  test    rdx, rsi #w flags
  adc     r10, rcx #r/w flags
  test    rdx, rsi #w flags
  adc     r10, rcx #r/w flags
  ...
  inc     i
  cmp     i, N
  jl      loop
```



Solution: Interleave a dependency breaking instruction

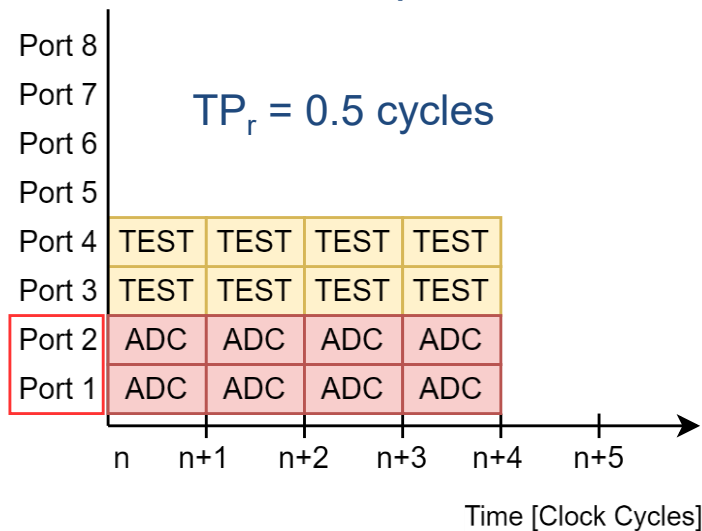
Does this affect runtime?

Microbenchmarking – Throughput

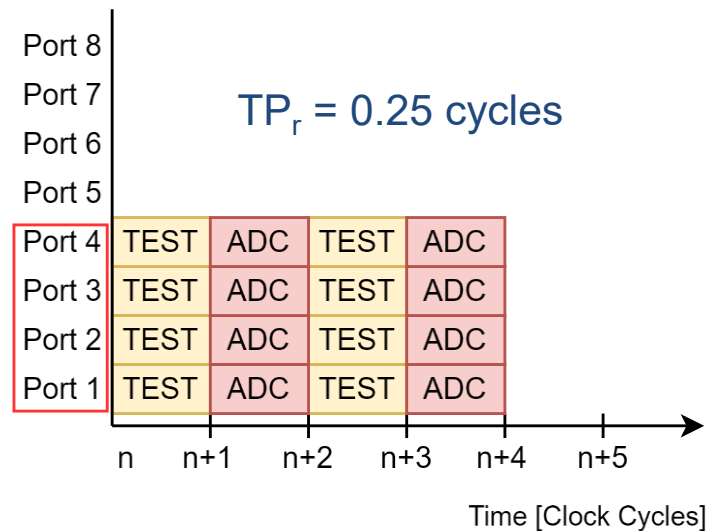
Do interleaved instructions affect the runtime?

$$TP_{r, \text{measured}} = 0.5 \text{ cycles}$$

Distinct ports



Shared ports



$$TP_{r, \text{ADC}} = [0.25; 0.5] \text{ cycles}$$

Microbenchmarking – Throughput

- Repeat instruction, choose different registers for r/w operands
- Implicit dependencies might prevent parallel execution
 - Break dependencies by interleaving other instructions
 - Causes uncertainties in the results

Microbenchmarking – Latency

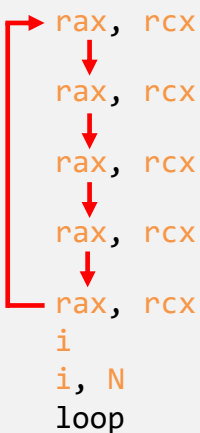
- Repeat instruction, build dependency chain

Bit test and set

BTS r64 (r/w) , r64 (r) , flags (w)

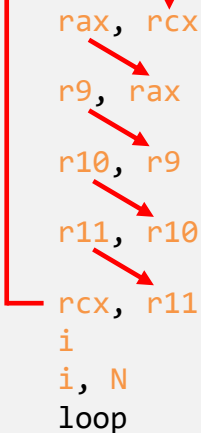
loop:

```
bts    rax, rcx
bts    rax, rcx
bts    rax, rcx
bts    rax, rcx
bts    rax, rcx
inc     i
cmp     i, N
jnl     loop
```



loop:

```
bts    rax, rcx
bts    r9, rax
bts    r10, r9
bts    r11, r10
bts    rcx, r11
inc     i
cmp     i, N
jnl     loop
```



It can make a
difference which
operands cause the
dependency!

Microbenchmarking – Latency

- Repeat instruction, build dependency chain

BTS r64 (r/w) , r64 (r) , flags (w)

```
loop: $ winic LAT -f 1.5 -i BTS64rr
      BTS64rr(1<GR64> -> 1<GR64>) [2;2]
      BTS64rr(2<GR64> -> 1<GR64>) [2;2]
      BTS64rr(1<GR64> -> 3<EFLAGS>) [ERR_NO_HELPER]
      BTS64rr(2<GR64> -> 3<EFLAGS>) [ERR_NO_HELPER]
```

bts rax, rcx
inc i
cmp i, N
jl loop

bts rcx, r11
inc i
cmp i, N
jl loop

difference which
operands cause the
dependency!

Microbenchmarking – Latency

- Repeat instruction, build dependency chain

```
BTS r64 (r/w) , r64 (r) , flags (w)  
SBB r64 (r/w) , r64 (r) , flags (r/w)
```

```
loop:  
  bts    rax, rcx #w flags  
  sbb    rdx, rsi #r flags  
  bts    r9, rdx  #w flags  
  sbb    r10, rsi #r flags  
  bts    r11, r10 #w flags  
  sbb    rcx, rsi #r flags  
  inc    i  
  cmp    i, N  
  jl     loop
```



Microbenchmarking – Latency

- Repeat instruction, build dependency chain

```
BTS r64(r/w), r64(r), flags(w)
```

```
SBB r64(r/r), r64(r), flags(r/r)
```

```
loop: $ winic LAT -f 1.5 -i BTS64rr SBB64rr
```

```
bt BTS64rr(1<GR64> -> 1<GR64>) [2;2]
```

```
sb BTS64rr(2<GR64> -> 1<GR64>) [2;2]
```

```
bt BTS64rr(1<GR64> -> 3<EFLAGS>) [1;1]
```

```
sb BTS64rr(2<GR64> -> 3<EFLAGS>) [1;1]
```

```
bt ...
```

```
sbb rcx, rsi #r flags
```

```
inc i
```

```
cmp i, N
```

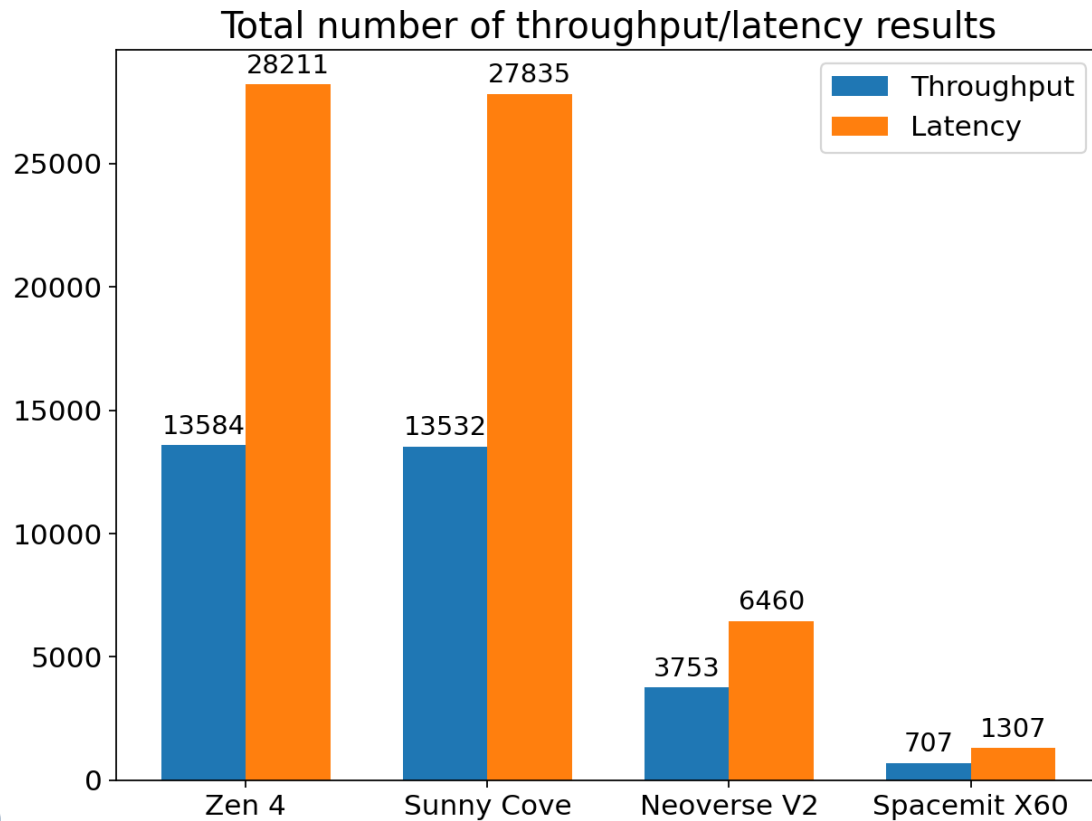
```
j1 loop
```

Different
Latency

Microbenchmarking – Latency

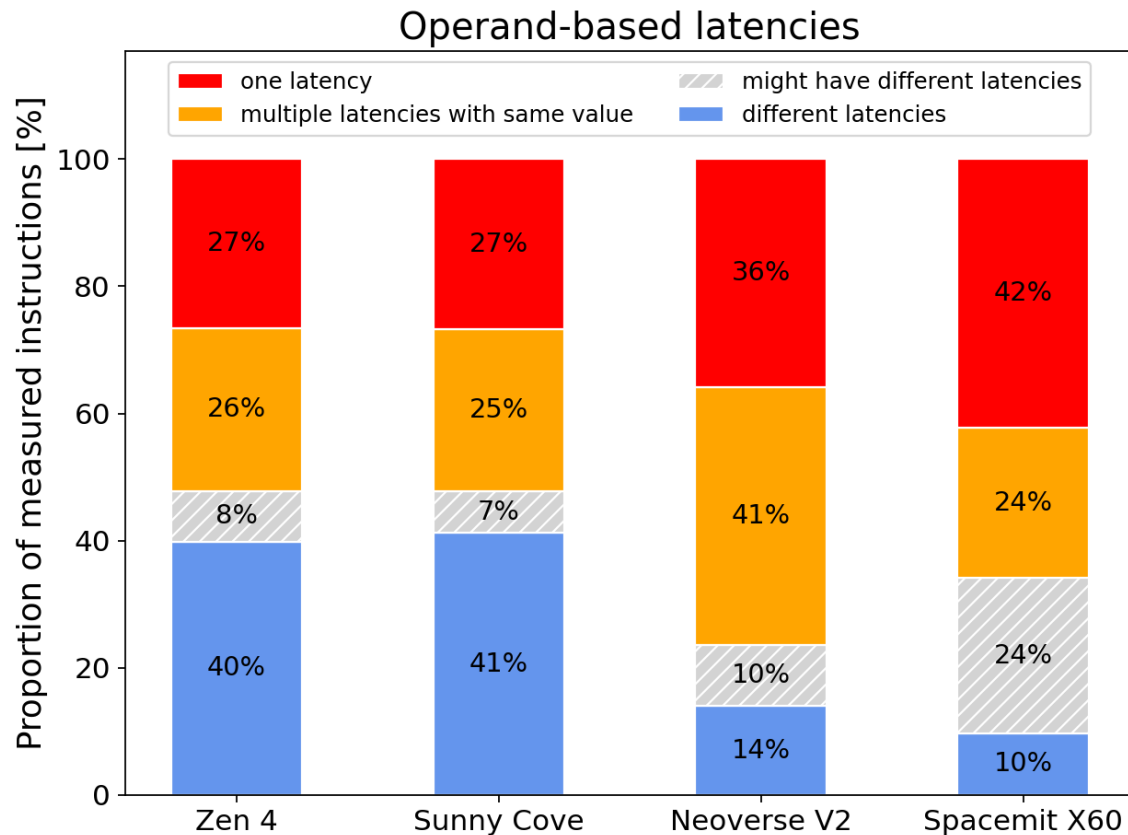
- Repeat instruction, build dependency chain
- Measure all operand-based latencies if possible
 - Interleave other instructions if necessary
 - May cause uncertainties

Results



x86 systems: AVX512
Neoverse V2: SVE2
Spacemit X60: RVV 1.0

Results – Latency



Results – Comparison to other sources

uops.info

llvm-exegesis

OSACA

documentation

Zen 4

Sunny Cove

Neoverse V2

Spacemit X60

```
{1,2} <-> {1} -> Full Match  
{1,2} <-> {1,3} -> Partial Match  
{1,2} <-> {3,4} -> No Match
```



Full Match



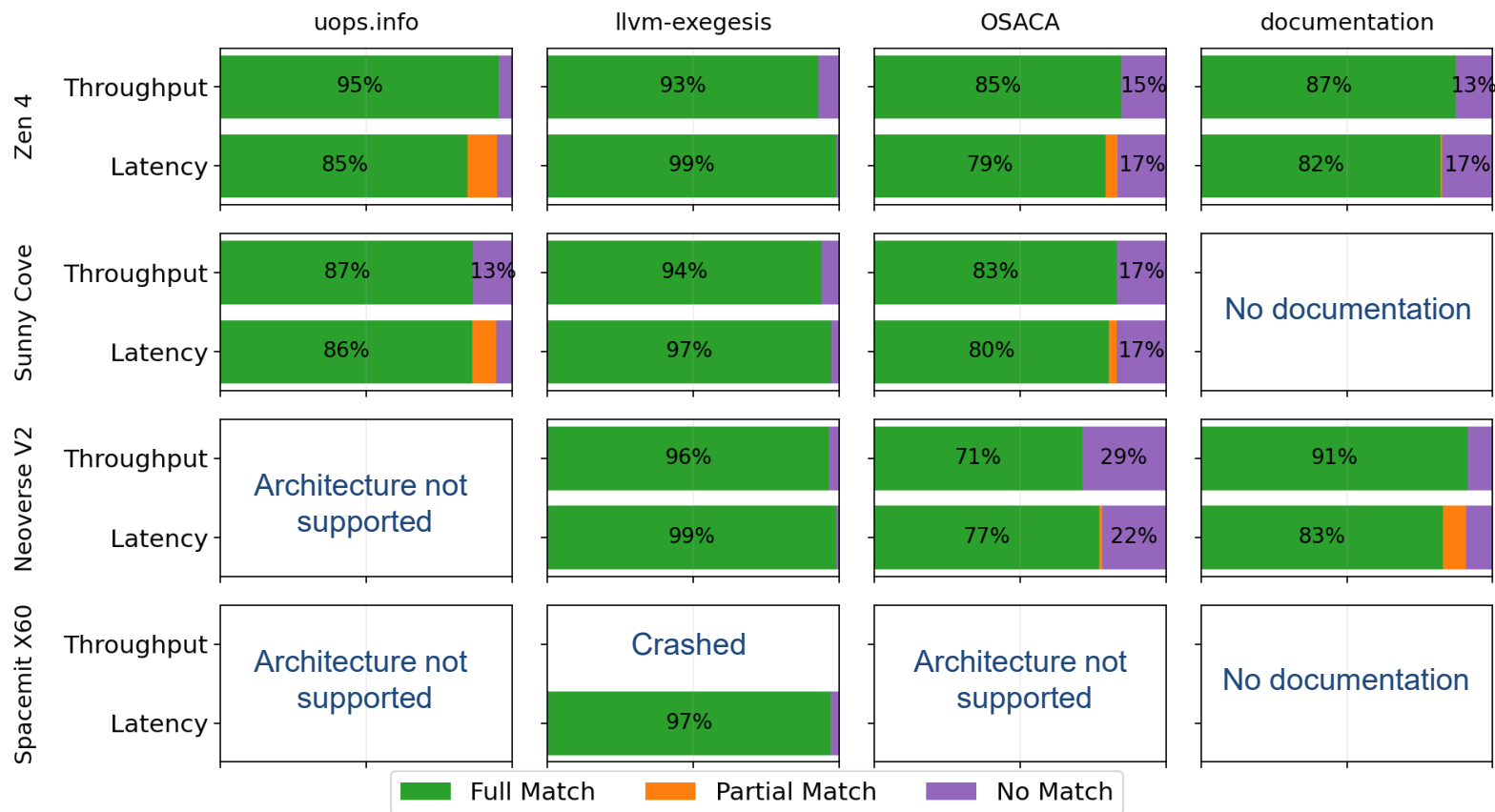
Partial Match



No Match

LLVM v22.1.8; OSACA 0.7.1; uops.info database version March 2026; V2 Optimization guide [4]; AMD optimization guide [10]

Results – Comparison to other sources



LLVM v22.1.8; OSACA 0.7.1; uops.info database version March 2026; V2 Optimization guide [4]; AMD optimization guide [10]

Results – Case Study

When do operand based latencies have an impact?

```
double a[N], b[N], c[N];
for(int i=1; i<N; ++i) {
    a[i] = a[i-1] + b[i] * c[i];
}
```

A variant of the Schönauer-DAXPY-
kernel with a read-after-write
dependency

```
.LBB0_2:
    ldr    d1, [x9], #8
    ldr    d2, [x10], #8
    subs   x11, x11, #1
    fmadd   d0, d1, d2, d0
    str    d0, [x8], #8
    b.ne   .LBB0_2
```

armv8-a clang 20.1.0
-mcpu=grace -O3 -fno-unroll-loops

Results – Case Study



When do operand based latencies have an impact?

```
$ osaca --arch v2 s_daxpy.s
```

	<ports>	LCD	
10			.LBB0_2:
11			ldr d1, [x9], #8
12			ldr d2, [x10], #8
13	<throughput analysis>		subs x11, x11, #1
14		4.0	fmadd d0, d1, d2, d0
15			str d0, [x8], #8
16			b.ne .LBB0_2
	1.0	4.0	

Throughput prediction Loop-carried dependency analysis

Results – Case Study



```
$ winic LAT -f 3.2 -i FMADDDrrr
FMADDDrrr(2<FPR64> -> 1<FPR64>) [4.01;4.01]
FMADDDrrr(3<FPR64> -> 1<FPR64>) [4.01;4.01]
FMADDDrrr(4<FPR64> -> 1<FPR64>) [2.01;2.01]
FMADDDrrr(Reg<FPCR> -> 1<FPR64>) [ERR_NO_HELPER]
```

```
12 | | | | ldr    d2, [x10], #8
13 | <throughput analysis> | | | subs   x11, x11, #1
14 | | | | 4.0 | fmadd   d0, d1, d2, d0
15 | | | | | | str    d0, [x8], #8
16 | | | | | | b.ne   .LBB0_2
    | 1.0 | 4.0 |
```

Measurement shows 2.0 cycles / iteration!

Results – Case Study



When do operand based latencies have an impact?

```
$ osaca --arch v2 s_daxpy.s
```

	<ports>	LCD	
10			.LBB0_2:
11			ldr d1, [x9], #8
12			ldr d2, [x10], #8
13	<throughput analysis>		subs x11, x11, #1
14		2.0	fmadd d0, d1, d2, d0
15			str d0, [x8], #8
16			b.ne .LBB0_2
	1.0	2.0	

Measurement shows 2.0 cycles / iteration!

Results – Limitations / Future Work

- Limitations
 - Privileged instructions
 - Branches, calls, returns
- Future Work
 - Operand-based latencies on memory access base registers
 - Port mapping
 - Gather and Scatter instructions

Summary

WINIC is an automatic multi-platform microbenchmarking tool

- Enables cycle-accurate in-core performance modeling
- No PMU access necessary

Supported metrics:

- throughput (with dependency breaking)
- operand-based latency

Platforms:

- x86
- AArch64
- RISC-V

Code and Results



 [github.com/
RRZE-HPC/WINIC](https://github.com/RRZE-HPC/WINIC)

References

- [1] <https://github.com/RRZE-HPC/OSACA> (OSACA)
- [2] <https://llvm.org/docs/CommandGuide/llvm-mca.html> (llvm-mca)
- [3] <https://dl.acm.org/doi/pdf/10.1145/3524059.3532396> (uiCA)
- [4] <https://developer.arm.com/documentation/109898/latest/> (Neoverse V2 Software Optimisation Guide)
- [5] <https://arxiv.org/abs/1911.03282> (NanoBench)
- [6] <https://github.com/RRZE-HPC/ibench> (iBench)
- [7] <https://github.com/RRZE-HPC/asmbench> (asmbench)
- [8] <https://llvm.org/docs/CommandGuide/llvm-exegesis.html> (llvm-exegesis)
- [9] <https://ieeexplore.ieee.org/document/7116904> (CQA)
- [10] <https://docs.amd.com/v/u/en-US/57647> (Software Optimization Guide for AMD Zen 4)

Results – Another Case Study

When do operand based latencies have an impact?

```
int NI, NK;
double a[NI][NK], b[NI][NK];

for (int i=1; i<NI-1; ++i) {
    for (int k=1; k<NK-1; ++k) {
        a[i][k] = (
            a[i+1][k]
            + a[i][k-1] + a[i][k+1]
            + a[i-1][k]
        ) / b[i][k];
    }
}
```

Gauss-Seidel with a divide operation

Results – Another Case Study

```
$ osaca --arch v2 gauss_seidel_div.s
```



Port pressure in cycles

	0	1	2	...	7	8	- 8DV	9	10	- 10DV	11	12	13	14	15	16	CP	LCD	
41																			.LBB0_7:
42			0.17		0.17												1.0		add x7, x18, x6
43			0.17		0.17														add x19, x0, x6
44			0.17		0.17														add x20, x1, x6
45			0.25		0.25														add x5, x5, #2
46												0.33	0.33	0.33			6.0		ldp d1, d2, [x7, #-8]
47			0.08		0.09														add x7, x17, x6
48			0.17		0.17														add x6, x6, #16
49						0.25		0.25	0.25		0.25						2.0	2.0	fadd d0, d1, d0
50												0.33	0.33	0.33					ldp d1, d3, [x19]
51						0.25		0.25	0.25		0.25						2.0	2.0	fadd d0, d0, d1
52												0.33	0.33	0.33					ldur d1, [x7, #-8]
53						0.25		0.25	0.25		0.25						2.0	2.0	fadd d0, d0, d1
54												0.33	0.33	0.33					ldur d1, [x20, #-8]
55						0.50	2.50		0.50	2.50							12.0	12.0	fdiv d0, d0, d1
56												0.50	0.50		0.50	0.50			stur d0, [x19, #-8]
57						0.25		0.25	0.25		0.25						2.0	2.0	fadd d0, d2, d0
58												0.33	0.33	0.33					ldr d1, [x7]
59			0.00		0.00														add x7, x16, x5
60						0.00		0.50	0.00		0.50						2.0	2.0	fadd d0, d0, d3
61						0.00		0.50	0.00		0.50						2.0	2.0	fadd d0, d0, d1
62												0.00	0.00	1.00					ldr d1, [x20]
63						0.50	2.50		0.50	2.50							12.0	12.0	fdiv d0, d0, d1
64												0.50	0.50		0.50	0.50	0.0		str d0, [x19]
65			0.33		0.33														cmp x7, #1
66	0.50	0.50																	b.ne .LBB0_7
	0.50	0.50	1.33	...	1.34	2.00	5.00	2.00	2.00	5.00	2.00	2.67	2.67	2.67	1.00	1.00	43.0	36.0	

OSACA 0.7.1, Architecture: neoverse-v2
<https://godbolt.org/z/dYvWrrarov>

Measurement shows 28.2 cycles / iteration!

Results – Another Case Study

```
$ osaca --arch v2 gauss_seidel_div.s
```

Port pressure in cycles

| 0 | 1 |

41 |

42 |

43 |

44 |

45 |

46 |

47 |

48 |

49 |

50 |

51 |

52 |

53 |

54 |

55 |

56 |

57 |

58 |

59 |

60 |

61 |

62 |

63 |

64 |

65 |

66 |

```
$ winic -f 3.2 LAT -i FDIVDrr
FDIVDrr(1<FPR64> -> 0<FPR64>) [7.01;7.01]
FDIVDrr(2<FPR64> -> 0<FPR64>) [14.5;14.5]
```



```
7:
x7, x18, x6
x19, x0, x6
x20, x1, x6
x5, x5, #2
d1, d2, [x7, #-8]
x7, x17, x6
x6, x6, #16
d0, d1, d0
ldp d1, d3, [x19]
fadd d0, d0, d1
ldur d1, [x7, #-8]
fadd d0, d0, d1
ldur d1, [x20, #-8]
fdiv d0, d0, d1
stur d0, [x19, #-8]
fadd d0, d2, d0
ldr d1, [x7]
add x7, x16, x5
fadd d0, d0, d3
fadd d0, d0, d1
ldr d1, [x20]
fdiv d0, d0, d1
str d0, [x19]
cmp x7, #1
b.ne .LBB0_7
```

0.50 0.50 1.33 ... 1.34 2.00 5.00 2.00 2.00 5.00 2.00 2.67 2.67 2.67 1.00 1.00 43.0 28.0

Results – Another Case Study

```
$ osaca --arch v2 ga
```

```
Port pressure in cyc  
| 0 | 1 |
```

```
41 |  
42 |  
43 |  
44 |  
45 |  
46 |  
47 |  
48 |  
49 |  
50 |  
51 |  
52 |  
53 |  
54 |  
55 |  
56 |  
57 |  
58 |  
59 |  
60 |  
61 |  
62 |  
63 |  
64 |  
65 |  
66 | 0.50 | 0.50 |  
  
0.50 0.50
```

The forwarding logic in the V pipelines is optimized to provide optimal latency for instructions which are expected to commonly forward to one another. The effective latency of FP and ASIMD instructions as described in section 3 is increased by one cycle if the producer and consumer instructions are not part of the same forwarding region. These optimized forwarding regions are defined in the following table.

Table 4-1 Optimized forwarding regions

Region	Instruction Types	Notes
1	ASIMD/SVE ALU, ASIMD/SVE shift, ASIMD/scalar insert and move, ASIMD/SVE abs/cmp/max/min and the ASIMD miscellaneous instructions in table 3-18.	1
2	FP/ASIMD/SVE multiply, FP/ASIMD/SVE multiply-accumulate, FP compare, FP add/sub and the ASIMD miscellaneous instructions in table 3-18.	1,2,3
3	ASIMD/SVE Crypto and SHA1/SHA256	-
4	ASIMD/SVE AES, ASIMD/SVE polynomial multiply and all the instruction types in region 1.	1
5	ASIMD/SVE BFDOT and BFMMMLA instructions	-

Notes:

1. Reciprocal step and estimate instructions are excluded from this region.
2. ASIMD extract narrow, saturating instructions are excluded from this region.
3. ASIMD miscellaneous instructions can only be consumers of this region.

The following instructions are not a part of any region:

- FP div/sqrt
- FP convert and rounding instructions that do not write to general purpose registers



```
16 || CP | LCD |  
||  
|| 1.0 |  
||  
||  
|| 6.0 |  
||  
|| 2.0 | 2.0 |  
|| 2.0 | 2.0 |  
|| 2.0 | 2.0 |  
|| 12.0 | 8.0 |  
0.50 || 2.0 | 2.0 |  
||  
|| 2.0 | 2.0 |  
|| 12.0 | 8.0 |  
0.50 || 0.0 |  
||  
1.00 43.0 28.0
```

.LBB0_7:
add x7, x18, x6
add x19, x0, x6
add x20, x1, x6
add x5, x5, #2
ldp d1, d2, [x7, #-8]
add x7, x17, x6
add x6, x6, #16
fadd d0, d1, d0
ldp d1, d3, [x19]
fadd d0, d0, d1
ldur d1, [x7, #-8]
fadd d0, d0, d1
ldur d1, [x20, #-8]
fdiv d0, d0, d1
stur d0, [x19, #-8]
fadd d0, d2, d0
ldr d1, [x7]
add x7, x16, x5
fadd d0, d0, d3
fadd d0, d0, d1
ldr d1, [x20]
fdiv d0, d0, d1
str d0, [x19]
cmp x7, #1
b.ne .LBB0_7