Karlsruhe Institute of Technology

Chair for Programming Paradigms
Chair for Embedded Systems

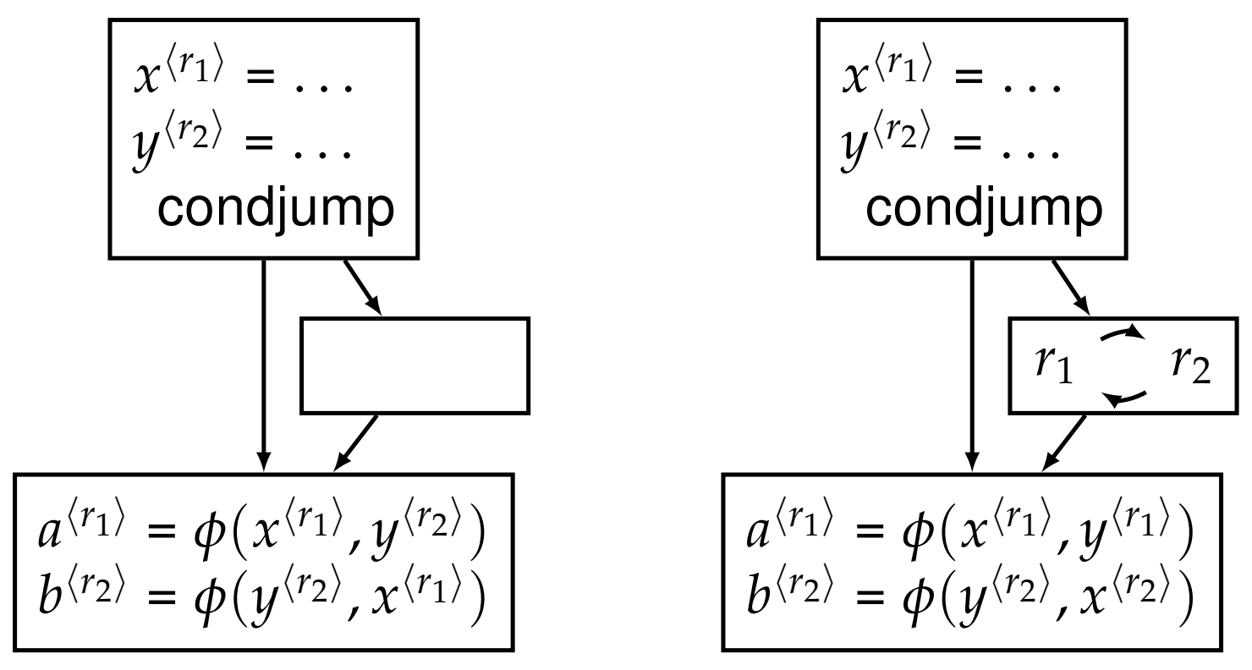
Hardware Acceleration for Programs in SSA Form

Manuel Mohr, Artjom Grudnitsky, Tobias Modschiedler, Lars Bauer, Sebastian Hack, Jörg Henkel

Introduction & Motivation

- Static Single Assignment (SSA) form has become key property of compiler intermediate languages
- Traditionally: SSA form destructed before register allocation
- Recent research: *SSA-based* register allocation

```
x = ...;
y = ...;
if (...) {
  t = x;
  x = y;
  y = t;
}
a = x;
b = y;
```

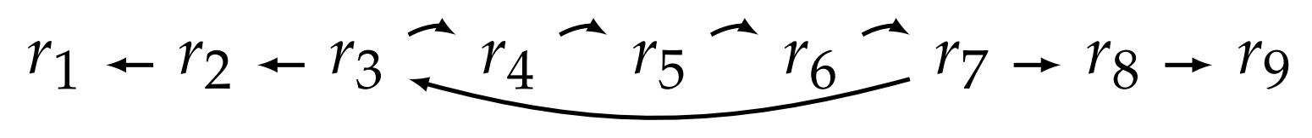

$x^{\langle r_1\rangle} = \ldots$
$y^{\langle r_2\rangle} = \ldots$
condjump
$a^{\langle r_1\rangle} = \phi(x^{\langle r_1\rangle}, y^{\langle r_2\rangle})$
$b^{\langle r_2\rangle} = \phi(y^{\langle r_2\rangle}, x^{\langle r_1\rangle})$
$x^{\langle r_1\rangle} = \ldots$
$y^{\langle r_2\rangle} = \ldots$
condjump
r_1 r_2
$a^{\langle r_1\rangle} = \phi(x^{\langle r_1\rangle}, y^{\langle r_1\rangle})$
$b^{\langle r_2\rangle} = \phi(y^{\langle r_2\rangle}, x^{\langle r_2\rangle})$

Shuffle Code

- ϕ-functions still present after register allocation

⇒ Must be implemented using *shuffle code*

- Shuffle code amount depends on copy coalescing quality



- On traditional machines: many instructions to implement
- Goal: Implement shuffle code in one instruction
- Fundamental hardware constraint: multiple write ports on register file extremely costly

⇒ Restriction to *register permutations*

Instruction Set Extension

Addition of permutation instructions to SPARC V8 ISA:

- 32 integer registers ⇒ 5 bits to identify one register
- 7 bits for opcode ⇒ 25 bits left for encoding 5 register numbers

31	27	24	21 19	14	9	4	0
0001	a_1	000	a_2	b	c	d	e

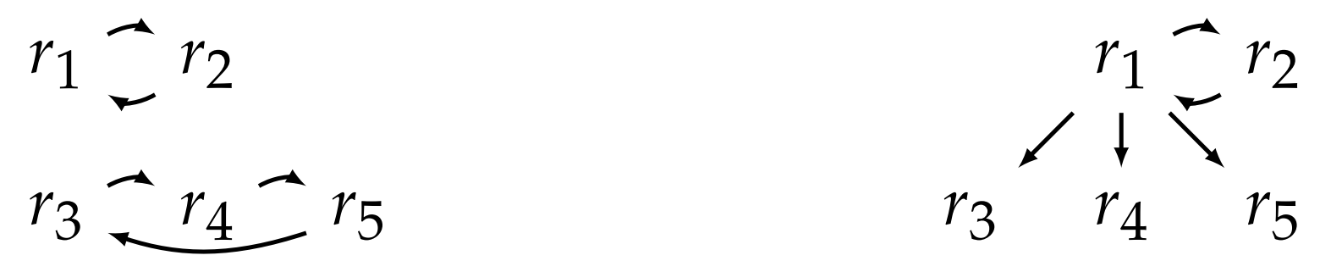
Two new instructions:

- `permi5`: Apply one cyclic permutation with *up to* 5 elements
- `permi23`: Apply two independent cycles with 2 and *up to* 3 elements

Code Generation

Register Transfer Graphs

- Directed graph $G = (V, E)$
- Each node $v \in V$ represents register
- Each edge (v, v') represents copy operation from v to v'
- Each node has at most one incoming edge
- All copy operations assumed to be performed *in parallel*

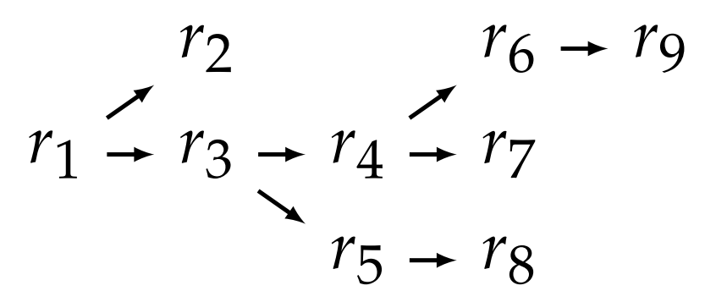


- RTGs only consisting of cycles (*permutation form*) can be implemented using only `permi` instructions
- In general: RTGs can duplicate values
- Permutations are injective, value duplication impossible

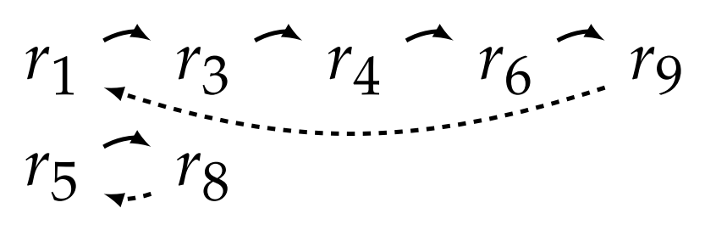
⇒ Two-phase approach to extract sub-RTG in permutation form

Phase 1: Conversion into Permutation Form

Input: Arbitrary RTG



Heuristics: At each node with > 1 outgoing edge:
Keep edge that is part of longest path starting at node
Output: RTG in permutation form + list of copy instructions



$+$

```
mov r3, r2
mov r6, r7
mov r4, r5
```

Phase 2: Decomposition into Cycles

Input: RTG in permutation form

Output: List of `permi` instructions that implement RTG

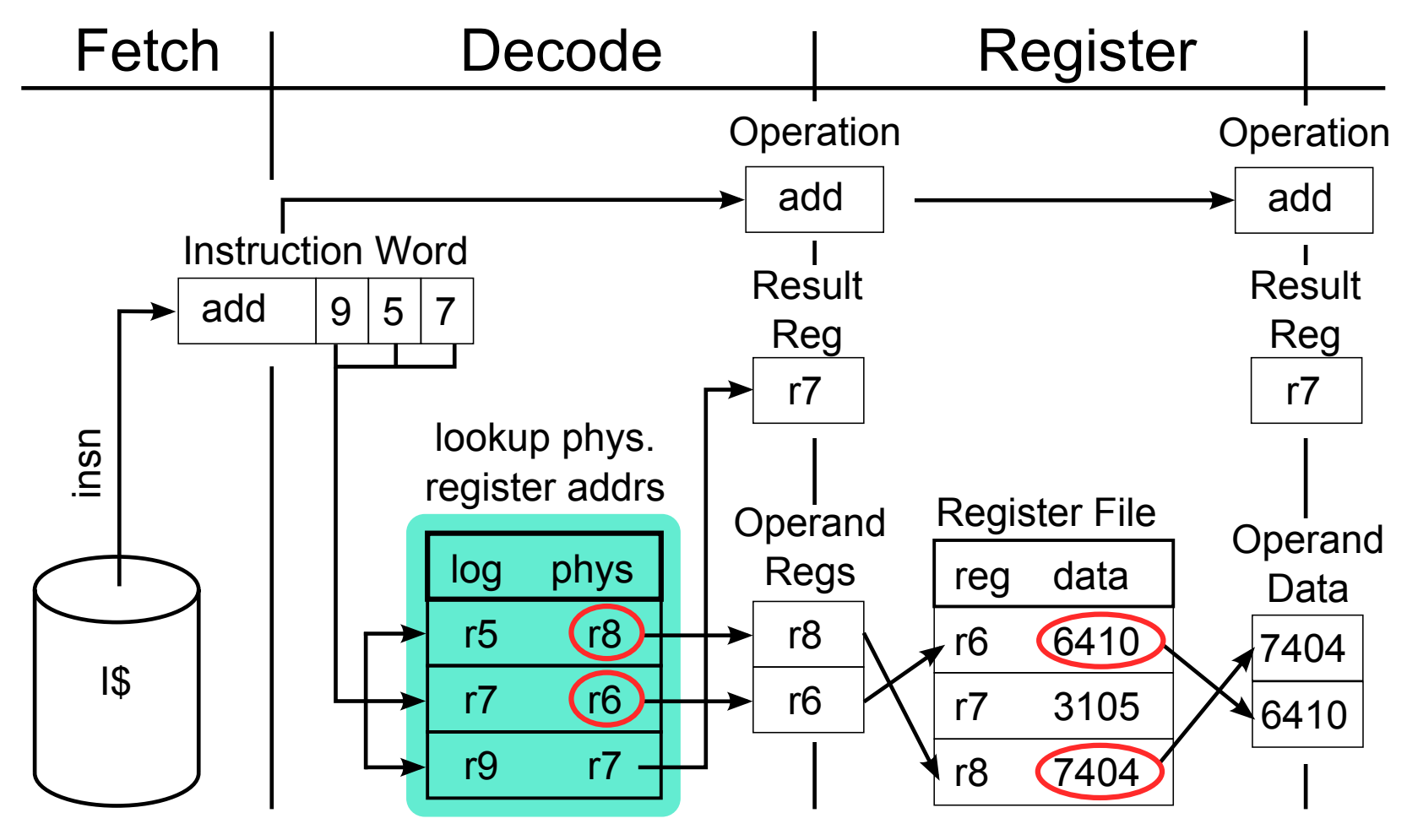
- Greedy algorithm with linear runtime shown on right

```
implementRegisterTransferGraph(rtg):
    insns $\leftarrow$ [] # List of generated instructions, initially empty
    (longs, shorts) $\leftarrow$ collectCycles(rtg)

    # First phase: only emit permi5 instructions
    while longs $\neq$ []:
        cycle $\leftarrow$ longs.take()
        while cycle.length() $\geq$ 4:
            (cycle', remainder) $\leftarrow$ split(cycle)
            insns.add(Permi5(cycle'))
            cycle $\leftarrow$ remainder
        if cycle.length() > 0:
            shorts.add(cycle) # Remember remainder

    # Second phase: try to fully utilize permi23 instructions
    (twos, threes) $\leftarrow$ sort(shorts)
    while (twos $\neq$ [] or threes $\neq$ []):
        if threes $\neq$ []:
            if twos $\neq$ []:
                insns.add(Permi23(twos.take(), threes.take()))
            else if threes.size() $\geq$ 2:
                (cycle2, cycle2') $\leftarrow$ split(threes.take())
                insns.add(Permi23(cycle2, threes.take()))
                twos.add(cycle2')
            else:
                insns.add(Permi5(threes.take()))
        else if twos $\neq$ []:
            if twos.size() $\geq$ 2:
                insns.add(Permi23(twos.take(), twos.take()))
            else:
                insns.add(Permi5(twos.take()))
```

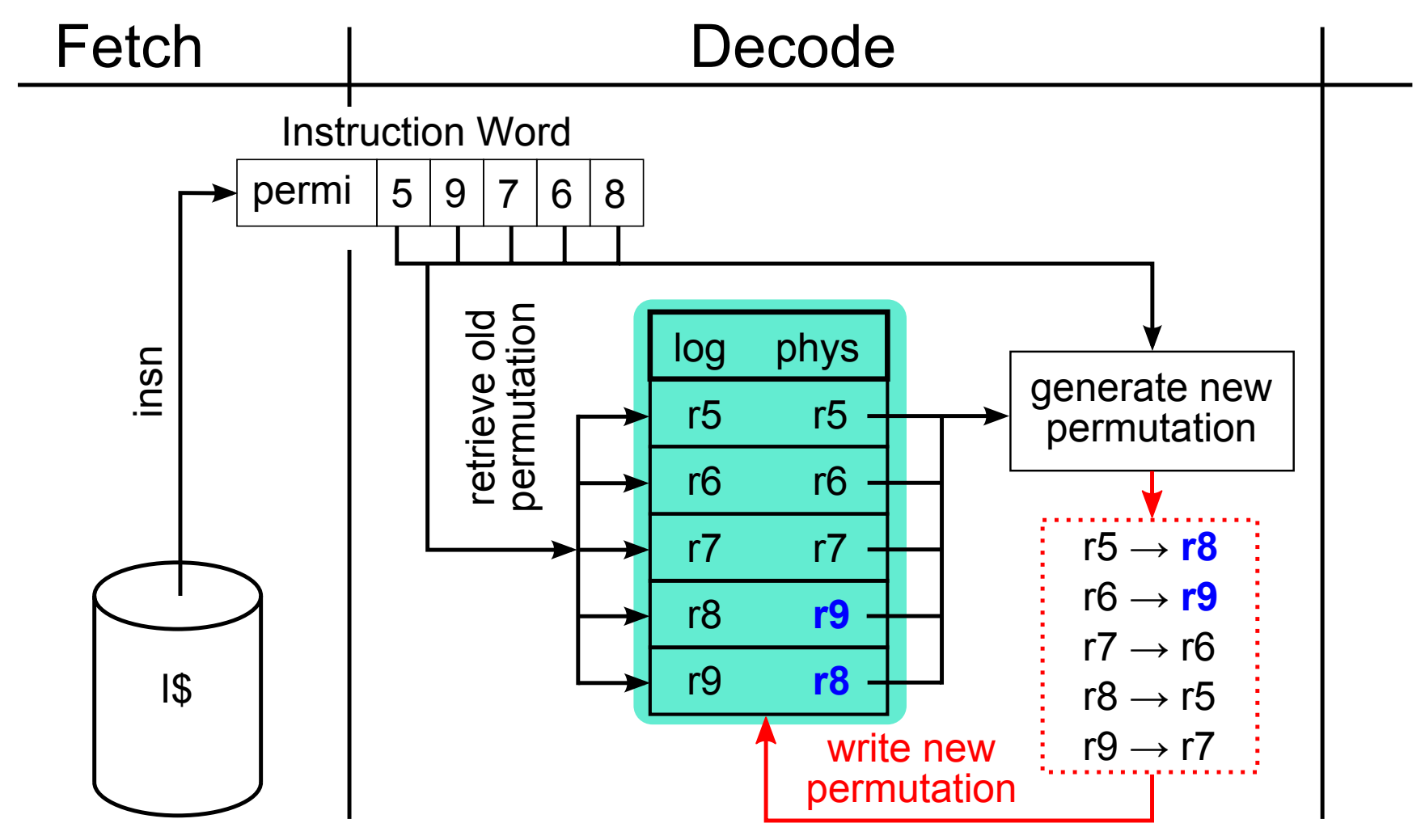
Hardware Implementation



`add` instruction going through pipeline stages

Key component: *permutation table* in Decode stage

- Contains mapping logical → physical register address
- Physical address used when accessing register file



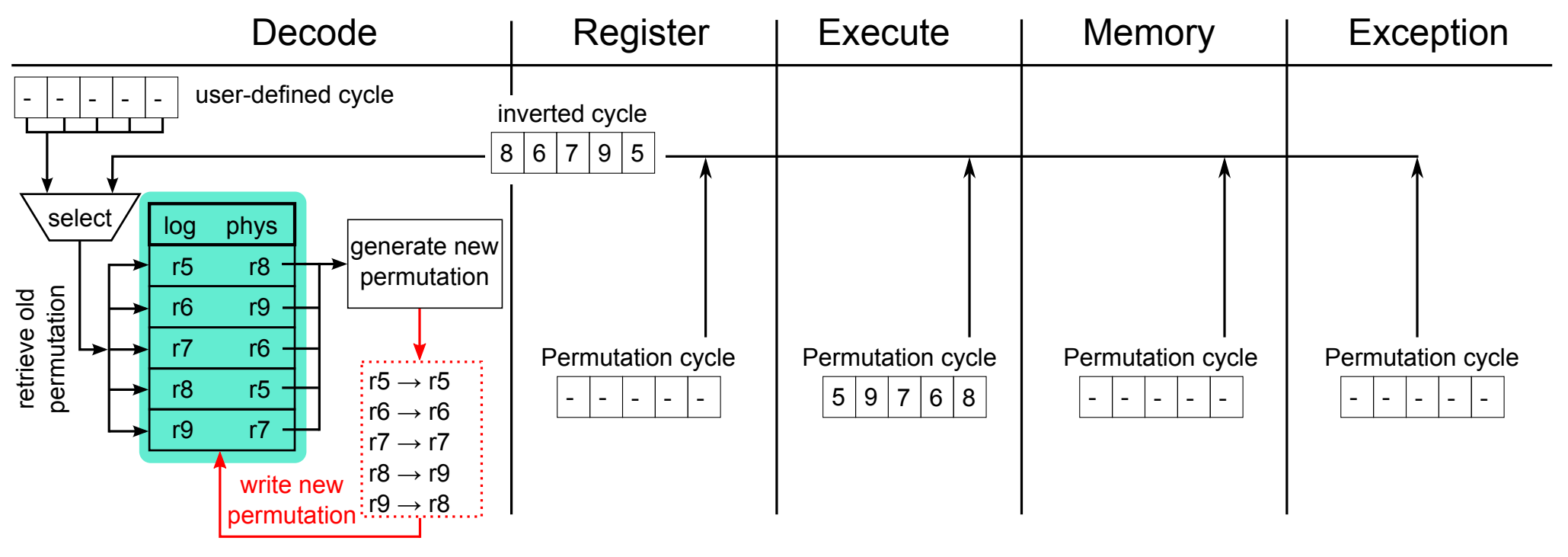
`permi` instruction going through pipeline stages

Applying permutation `permi5 r5 r9 r7 r6 r8`:

- Permutation performed in Decode stage (*early committing*)
- No changes to forwarding logic required

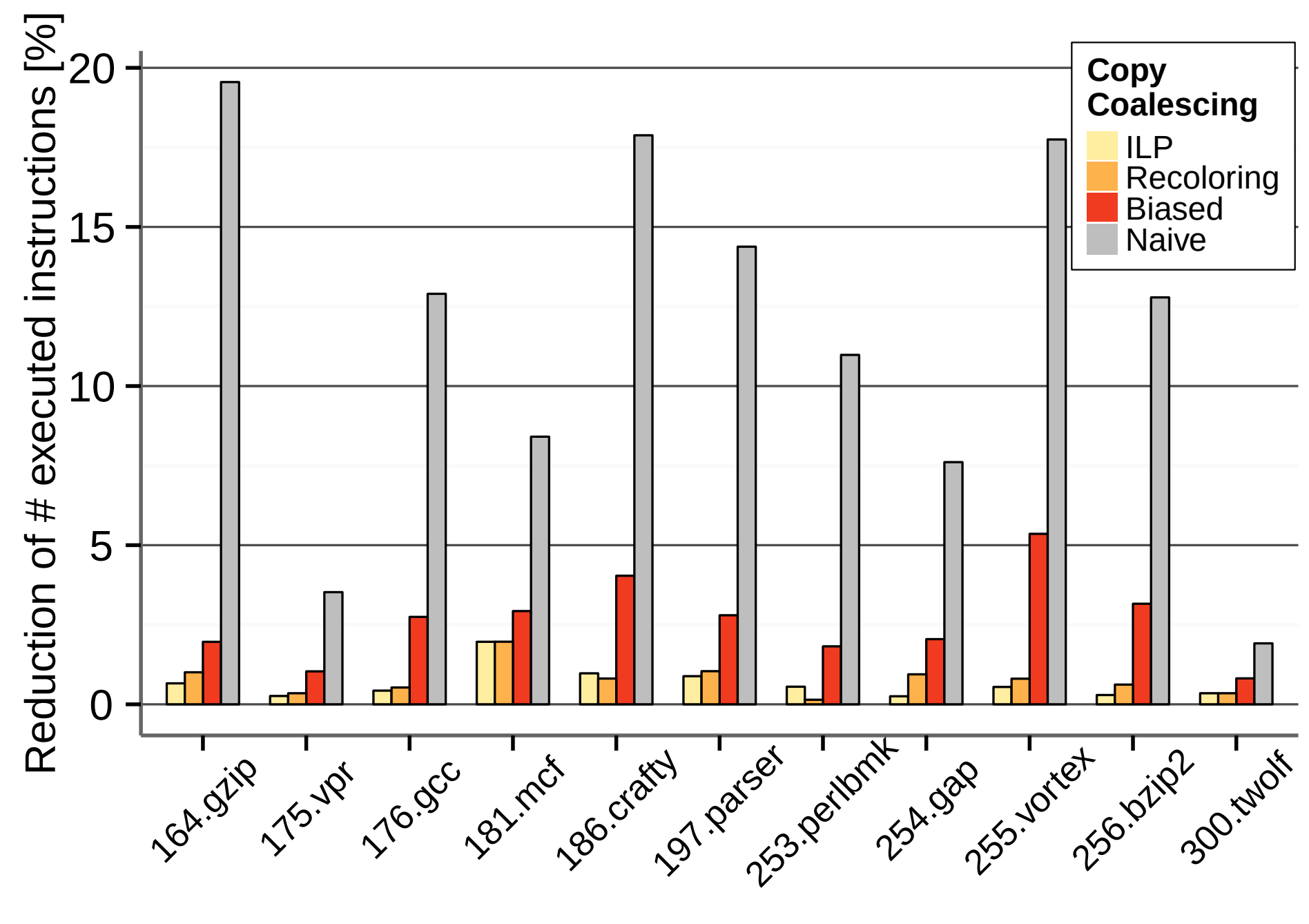
Permutations may need to be *reversed* if trap occurs (OS scheduler, I/O activity, etc.)

- Regular SPARC instructions: *annul* instructions
- *permi* instructions: check previous pipeline stages for `permi` instructions
  - If any: apply inverse permutations to permutation table

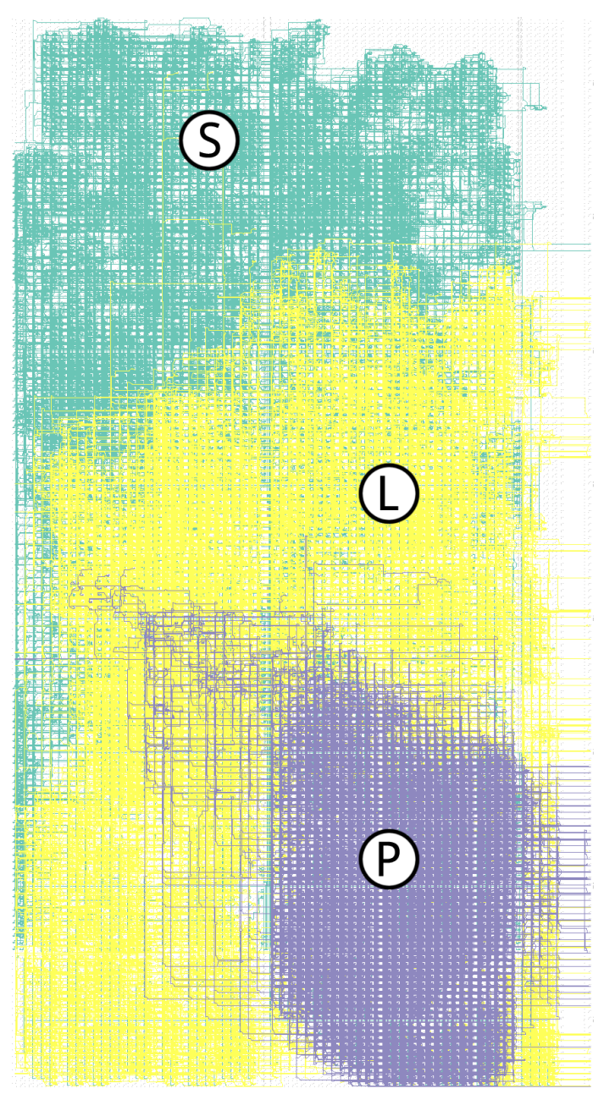


Pipeline snapshot during permutation reversion

Experimental Evaluation & Conclusion

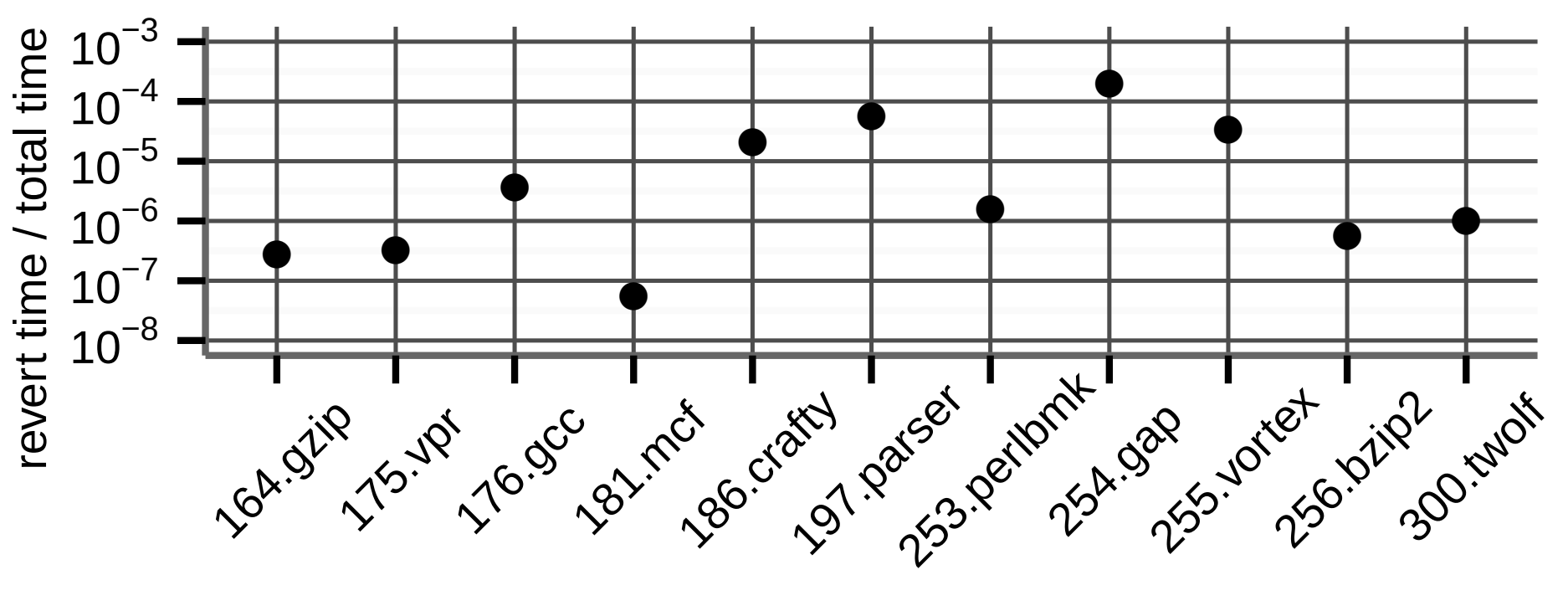


# instructions per RTG	SPARC	Our system	Reduction
ILP (best)	1.86	1.14	38.6%
Recolor	2.04	1.14	44.0%
Biased	2.99	1.54	48.5%
Naive (worst)	5.12	1.85	63.7%



Ⓛ – Gaisler LEON 3 Processor
Ⓟ – Permutator Extension
Ⓢ – Other SoC components (DDR Controller, Ethernet, Debug Unit, etc.)

FPGA Utilization	Base system	Our system	Overhead
LUTs	21%	31%	44%
Slices	41%	55%	31%
Flip-flops	11%	12%	16%
BlockRAMs	19%	19%	0%
Frequency	80 MHz	80 MHz	0%



Experimental Setup

- Implemented code generation strategy in libFirm compiler
- SPEC CPU2000 benchmark suite as input programs
- Modified QEMU to support `permi` instructions
- Ability to get precise dynamic instruction counts
- Hardware prototype based on Gaisler LEON 3 processor
- Measurements on hardware prototype implementation on Virtex-5 FPGA

Conclusion

- Novel approach to accelerate shuffle code by hardware extension
- New instructions added to standard instruction set
- Code generation approach producing efficient code fast
- Extensive evaluation including FPGA prototype implementation
- Universal speedup, number of executed instructions reduced by up to 5.1%

KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

www.kit.edu