Karlsruhe Institute of Technology

SAARLAND UNIVERSITY

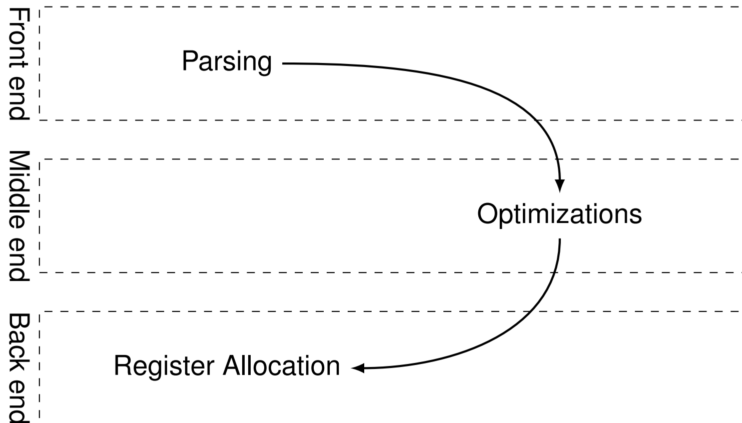
COMPUTER SCIENCE

Hardware Acceleration for Programs in SSA Form

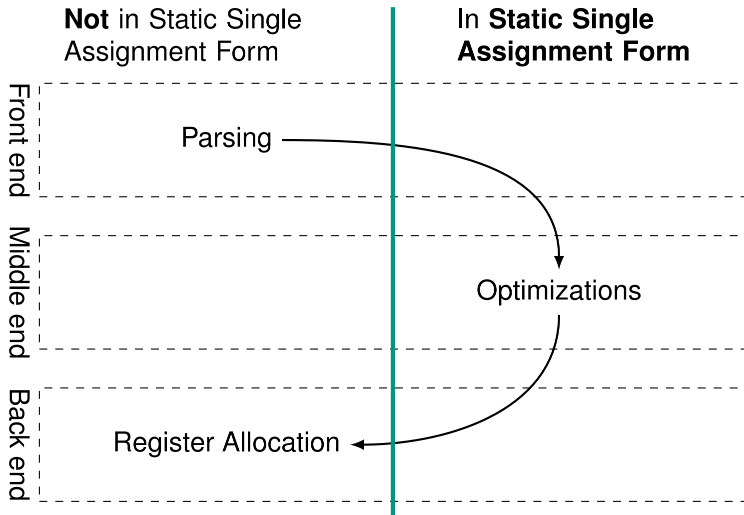
Manuel Mohr, Artjom Grudnitsky, Tobias Modschiedler, Lars Bauer, Sebastian Hack, Jörg Henkel

Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology (KIT)

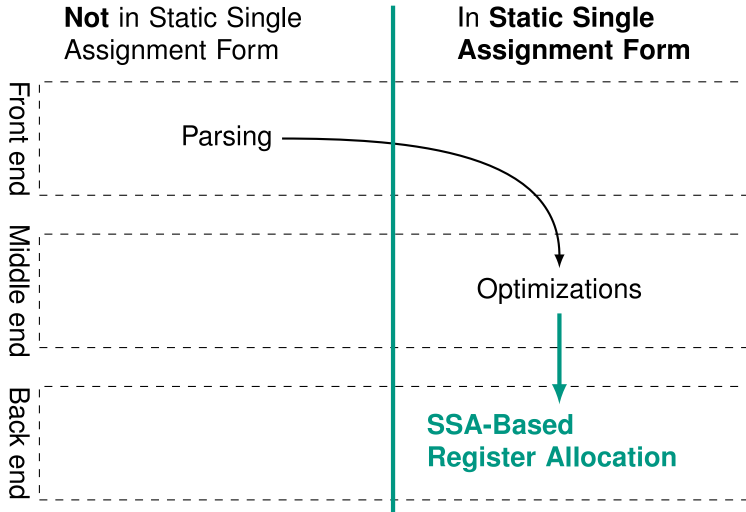
SSA-Based Register Allocation



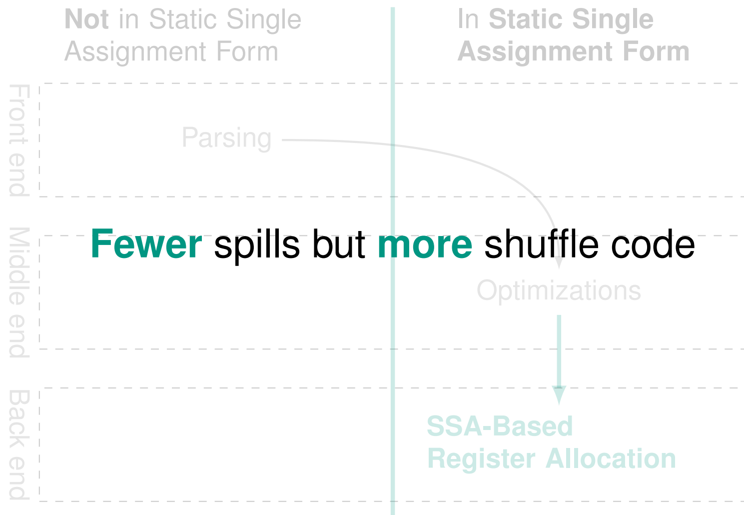
SSA-Based Register Allocation



SSA-Based Register Allocation



SSA-Based Register Allocation



Fewer spills but **more** shuffle code

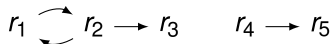
Register Transfer Graphs

Shuffle code = **parallel** copy operations between registers

Register Transfer Graphs

Karlsruhe Institute of Technology

Shuffle code = **parallel** copy operations between registers



Register Transfer Graph (RTG)

- Nodes: Registers
- Directed edge (r_1, r_2): After copies, value of r_1 must be in r_2
- At most one incoming edge per node
- No incoming edge: Register value is irrelevant after copies

Motivation

- Number and size of RTGs depend on quality of allocation
- Reduction is an NP-complete problem

⇒ On standard hardware, implementation may be expensive: 5% to 20% of all generated instructions (SPEC)

Motivation

- Number and size of RTGs depend on quality of allocation
- Reduction is an NP-complete problem

⇒ On standard hardware, implementation may be expensive:
5% to 20% of all generated instructions (SPEC)

```
mov r2, r1        xor r6, r7        xor r4, r5
mov r3, r2        xor r6, r5        xor r5, r4
mov r7, r8        xor r5, r6        xor r4, r3
xor r6, r7        xor r6, r5        xor r3, r4
xor r7, r6        xor r5, r4        xor r4, r3
```

Motivation

Karlsruhe Institute of Technology

- Number and size of RTGs depend on quality of allocation
- Reduction is an NP-complete problem

⇒ On standard hardware, implementation may be expensive:
5% to 20% of all generated instructions (SPEC)

Question 1: Is it possible to create an instruction set extension that allows implementing an RTG in one processor cycle?

Motivation

- Number and size of RTGs depend on quality of allocation
- Reduction is an NP-complete problem

⇒ On standard hardware, implementation may be expensive:
5% to 20% of all generated instructions (SPEC)

Question 1: Is it possible to create an instruction set extension that allows implementing an RTG in one processor cycle?

Question 2: Is it worth it?

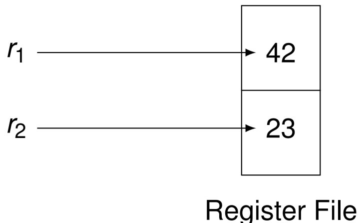
Fundamental Hardware Constraints

Karlsruhe Institute of Technology

- Changing contents of multiple registers in one cycle very costly

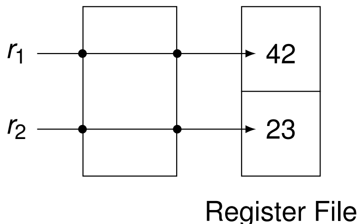
Fundamental Hardware Constraints

- Changing contents of multiple registers in one cycle very costly
- Idea: Modify *access* to register file instead of contents
  - Swap r_1 and r_2: Exchange the access to r_1 and r_2



Fundamental Hardware Constraints

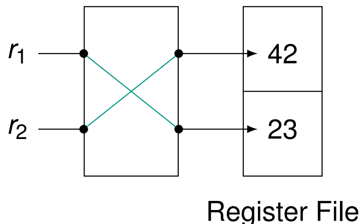
- Changing contents of multiple registers in one cycle very costly
- Idea: Modify *access* to register file instead of contents
  - Swap r_1 and r_2: Exchange the access to r_1 and r_2



Fundamental Hardware Constraints

Karlsruhe Institute of Technology

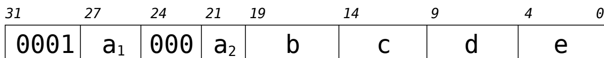
- Changing contents of multiple registers in one cycle very costly
- Idea: Modify *access* to register file instead of contents
  - Swap r_1 and r_2: Exchange the access to r_1 and r_2



⇒ Restriction to *permutations* of registers

ISA Extension

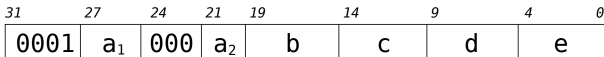
- Add permutation instructions to SPARC V8 ISA
- 32 registers ⇒ 5 bits to identify one register
- 7 bits for opcode ⇒ 25 bits left for encoding 5 register numbers



ISA Extension

Karlsruhe Institute of Technology

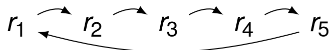
- Add permutation instructions to SPARC V8 ISA
- 32 registers $\Rightarrow$ 5 bits to identify one register
- 7 bits for opcode $\Rightarrow$ 25 bits left for encoding 5 register numbers



Two new instructions:

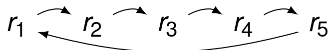
- `permi5`: Implement cyclic RTG with *up to* 5 elements
- `permi23`: Implement two independent cycles with 2 and *up to* 3 elements

Examples




```
permi5 r1, r2, r3, r4, r5
```

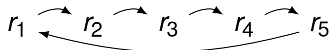
Examples




```
permi5 r1, r2, r3, r4, r5
```

```
permi5 r1, r2
```

Examples



`permi5 r1, r2, r3, r4, r5`

`permi5 r1, r2`

`permi23 r1, r2, r3, r4`

Code Generation

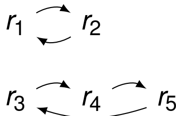
Karlsruhe Institute of Technology

- Goal: Generate efficient code using `permi` instructions for all RTGs
- Question: Which RTGs can be implemented using only `permi`?

Code Generation

- Goal: Generate efficient code using `permi` instructions for all RTGs
- Question: Which RTGs can be implemented using only `permi`?

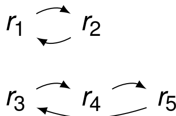
- RTGs in permutation form
  - Permutation can be written as a product of cycles
  - Cycles can be implemented with `permi`s



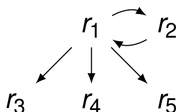
Code Generation

- Goal: Generate efficient code using `permi` instructions for all RTGs
- Question: Which RTGs can be implemented using only `permi`?

- RTGs in permutation form
  - Permutation can be written as a product of cycles
  - Cycles can be implemented with `permi`s

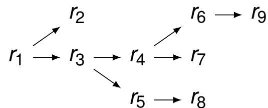


- In general: RTGs can duplicate values
  - Permutations are injective
  - Value duplication impossible



Two-Phase Approach

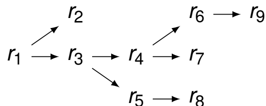
Arbitrary RTG



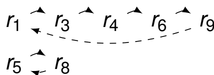
Two-Phase Approach

Karlsruhe Institute of Technology

Arbitrary RTG



Phase 1:
Conversion



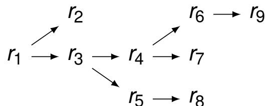
$+$

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

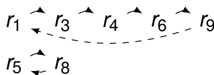
Two-Phase Approach

Karlsruhe Institute of Technology

Arbitrary RTG



Phase 1:
Conversion



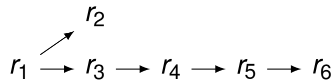
+

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

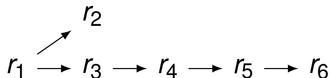
Phase 2:
Decomposition

```
permi5 r1, r3, r4, r6, r9
permi5 r5, r8
```

Conversion into Permutation Form

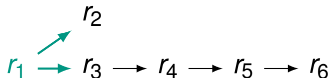


Conversion into Permutation Form



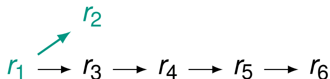
- At each node with >1 outgoing edge: keep edge that is part of longest path starting at node

Conversion into Permutation Form



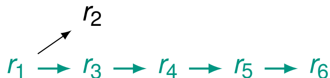
- At each node with > 1 outgoing edge: keep edge that is part of longest path starting at node

Conversion into Permutation Form



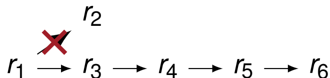
- At each node with >1 outgoing edge: keep edge that is part of longest path starting at node

Conversion into Permutation Form



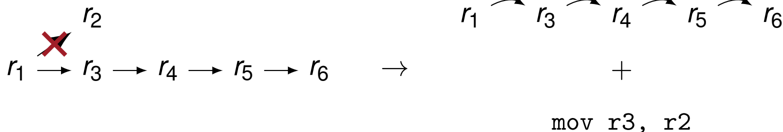
- At each node with >1 outgoing edge: keep edge that is part of longest path starting at node

Conversion into Permutation Form



- At each node with >1 outgoing edge: keep edge that is part of longest path starting at node

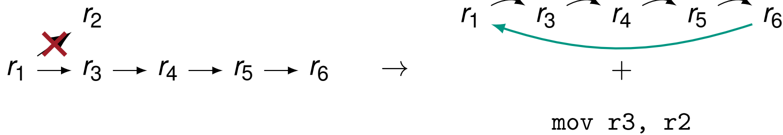
Conversion into Permutation Form



- At each node with $>$ 1 outgoing edge: keep edge that is part of longest path starting at node

Conversion into Permutation Form

Karlsruhe Institute of Technology



- At each node with >1 outgoing edge: keep edge that is part of longest path starting at node

Decomposition into Cycles

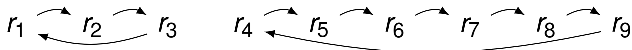
Karlsruhe Institute of Technology

- After conversion: Implement RTG in permutation form with as few `permi`s as possible
- Need to combine multiple cycles to exploit `permi23`

Decomposition into Cycles

Karlsruhe Institute of Technology

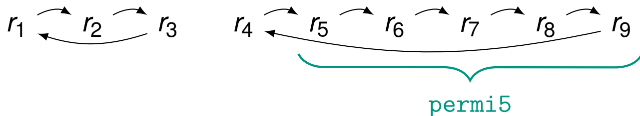
- After conversion: Implement RTG in permutation form with as few `permi`s as possible
- Need to combine multiple cycles to exploit `permi23`



Decomposition into Cycles

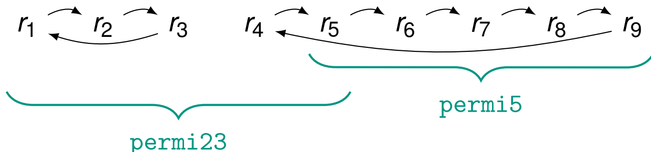
Karlsruhe Institute of Technology

- After conversion: Implement RTG in permutation form with as few `permi`s as possible
- Need to combine multiple cycles to exploit `permi23`



Decomposition into Cycles

- After conversion: Implement RTG in permutation form with as few `permi`s as possible
- Need to combine multiple cycles to exploit `permi23`



Decomposition into Cycles

Karlsruhe Institute of Technology

- Greedy decomposition algorithm with linear runtime

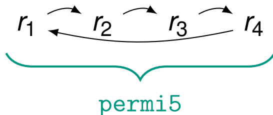
- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it

Decomposition into Cycles

Karlsruhe Institute of Technology

- Greedy decomposition algorithm with linear runtime

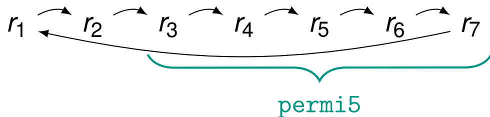
- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it



Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it



Decomposition into Cycles

Karlsruhe Institute of Technology

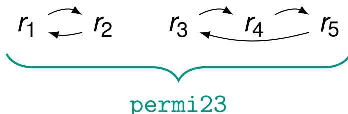
- Greedy decomposition algorithm with linear runtime

- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it
- **Phase 2**: Only cycles of size ≤ 3 left

Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

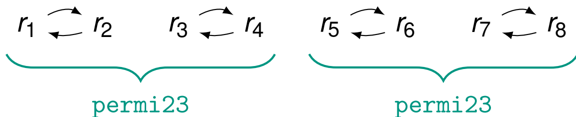
- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it
- **Phase 2**: Only cycles of size $\leq$ 3 left
  - If 2-cycle and 3-cycle available: combine using `permi23`



Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

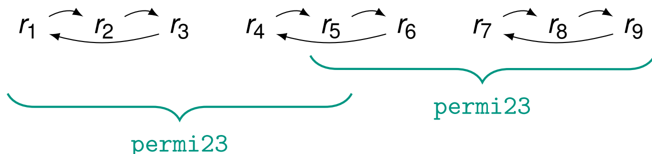
- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it
- **Phase 2**: Only cycles of size $\leq$ 3 left
  - If 2-cycle and 3-cycle available: combine using `permi23`
  - If only 2-cycles available: combine in pairs using `permi23`



Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

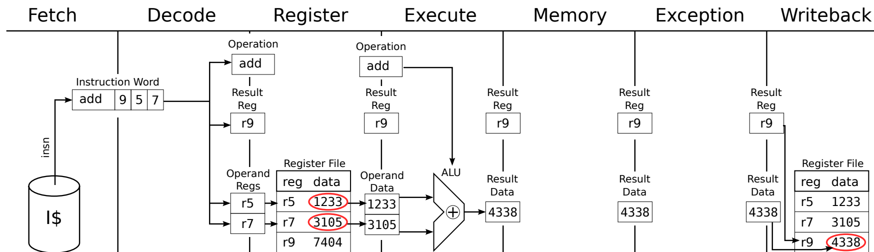
- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it
- **Phase 2**: Only cycles of size ≤ 3 left
  - If 2-cycle and 3-cycle available: combine using `permi23`
  - If only 2-cycles available: combine in pairs using `permi23`
  - If only 3-cycles available: combine in groups of three using `permi23`



Base Architecture

Karlsruhe Institute of Technology

- Underlying architecture: Gaisler LEON3, 7-stage pipeline
- Example: `add r9 r5 r7`

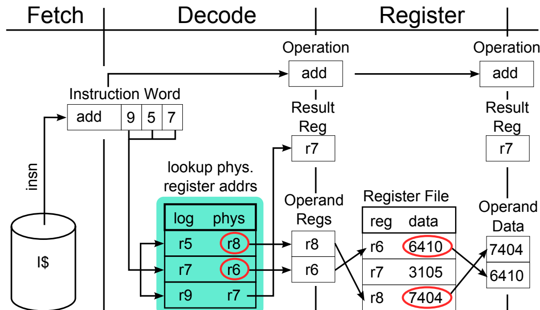


- For `permi` support: modifications of **Decode** and **Exception** stages

Manuel Mohr, Artjom Grudnitsky, Tobias Modschiedler, Lars Bauer, Sebastian Hack, Jörg Henkel – Hardware Acceleration for Programs in SSA Form
IPD, ITEC

Permutation Support

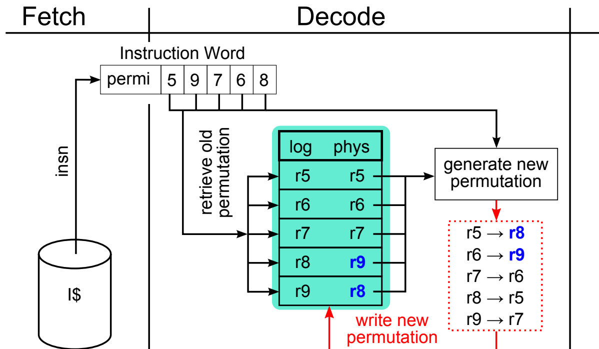
- Key component: permutation table in Decode stage
  - Contains mapping logical → physical register address
  - Physical address from permutation table used when accessing register file
- Initialized with *identity* at system reset



Applying new Permutations

Karlsruhe Institute of Technology

- Applying permutation `permi5 r5 r9 r7 r6 r8`



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

Results

Karlsruhe Institute of Technology

Experimental evaluation

- Implemented code generation strategy in libFIRM
- Used SPEC CPU2000 benchmark suite as input programs
- Modified SPARC emulator to support `permi` instructions
  - Ability to get precise dynamic instruction counts
- Validation by measurements on FPGA prototype implementation
  - By running Linux on FPGA prototype, ability to reuse executables

Compile Time

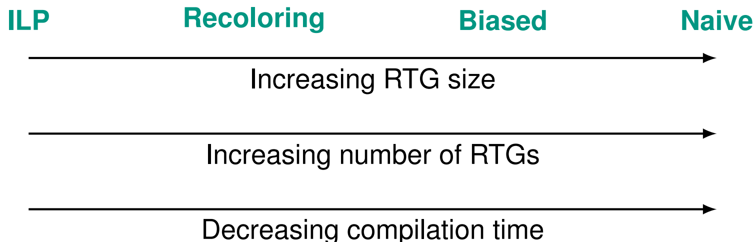
Karlsruhe Institute of Technology

	Default [ms]	Our code gen. [ms]	Relative
Backend (total)	63 598.0	63 927.0	**+0.5%**

- Code generation does not cause significant overhead

Code Quality

Four different register allocator configurations:



Code Quality

Karlsruhe Institute of Technology

Benchmark	**ILP**	**Recoloring**	**Biased**	**Naive**
164.gzip	−0.7%	−1.0%	−1.9%	**−16.4%**
175.vpr	−0.3%	−0.3%	−1.0%	−3.4%
176.gcc	−0.4%	−0.5%	−2.7%	−11.4%
181.mcf	**−1.9%**	**−1.9%**	−2.8%	−7.8%
186.crafty	−1.0%	−0.8%	−3.9%	−15.2%
197.parser	−0.9%	−1.0%	−2.7%	−12.6%
253.perlbmk	−0.6%	−0.1%	−1.8%	−9.9%
254.gap	−0.3%	−0.9%	−2.0%	−7.1%
255.vortex	−0.5%	−0.8%	**−5.1%**	−15.1%
256.bzip2	−0.3%	−0.6%	−3.1%	−11.3%
300.twolf	−0.3%	−0.3%	−0.8%	−1.9%

- Relative change of number of executed instructions

Code Quality

Karlsruhe Institute of Technology

Benchmark	ILP	Recoloring	Biased	Naive
164.gzip	−0.7%	−1.0%	−1.9%	**−16.4%**
175.vpr	−0.3%	−0.3%	−1.0%	−3.4%
176.gcc	−0.4%	−0.5%	−2.7%	−11.4%
181.mcf	**−1.9%**	**−1.9%**	−2.8%	−7.8%
186.crafty	−1.0%	−0.8%	−3.9%	−15.2%
197.parser	−0.9%	−1.0%	−2.7%	−12.6%
253.perlbmk	−0.6%	−0.1%	−1.8%	−9.9%
254.gap	−0.3%	−0.9%	−2.0%	−7.1%
255.vortex	−0.5%	−0.8%	**−5.1%**	−15.1%
256.bzip2	−0.3%	−0.6%	−3.1%	−11.3%
300.twolf	−0.3%	−0.3%	−0.8%	−1.9%

- Relative change of number of executed instructions
- Universal reduction, up to 5.1% for realistic scenarios
- The worse the register allocation, the higher the benefit using `permis`
- Confirmation by FPGA measurements, speedup up to 1.07

Area Overhead

Karlsruhe Institute of Technology

	Base system	Our system	Overhead
Frequency	80 MHz	80 MHz	0%
BlockRAMs	28	28	0%
Flip-flops	7 607	8 851	**16%**
LUTs	15 024	21 630	**44%**
Slices	7 249	9 507	**31%**

- Frequency unaffected
- Logical-physical mapping ⇒ increase in FF usage
- Large multiplexers ⇒ increase in LUT usage
  - Considerably smaller overhead for ASIC implementation

Conclusion

Karlsruhe Institute of Technology

Summary

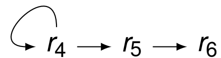
- Novel approach to accelerate shuffle code by hardware extension
- New instructions added to standard instruction set
- Code generation approach producing efficient code fast
- Extensive evaluation including FPGA prototype implementation
- Universal speedup, instruction count reduction up to 5.1%

Karlsruhe Institute of Technology

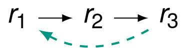
Backup Slides

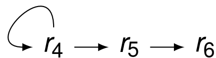
RTG Semantics

$r_1 \longrightarrow r_2 \longrightarrow r_3$



RTG Semantics

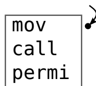




Exception Handling

- Early committing can cause problems due to traps
  - Timer interrupts to invoke OS scheduler
  - SPARC window overflows/underflows caused by nested function calls
- Trap handling in LEON3:

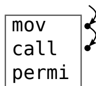
Fetch	Decode	Register	Execute	Memory	Exception	Writeback
-	-	-	permi	call	mov	-



Exception Handling

- Early committing can cause problems due to traps
  - Timer interrupts to invoke OS scheduler
  - SPARC window overflows/underflows caused by nested function calls
- Trap handling in LEON3:

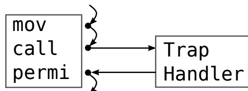
Fetch	Decode	Register	Execute	Memory	Exception	Writeback
-	-	-	-	permi	call ⚡	mov



Exception Handling

- Early committing can cause problems due to traps
  - Timer interrupts to invoke OS scheduler
  - SPARC window overflows/underflows caused by nested function calls
- Trap handling in LEON3:

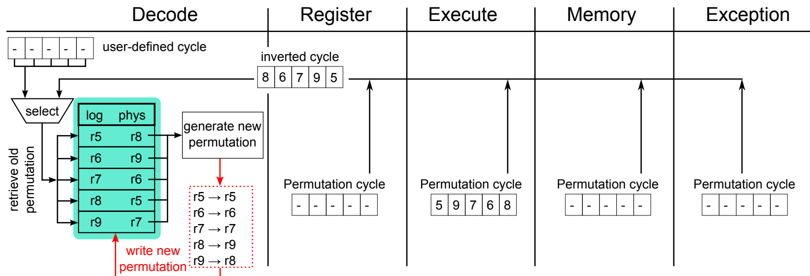
Fetch	Decode	Register	Execute	Memory	Exception	Writeback
permi	-	-	-	-	-	-



- permi executed twice – permutation applied twice $\rightarrow$ program crash
- Instructions that commit after exception stage can be annulled
- permi: revert effect of permutations executed before trap

Reverting Permutations

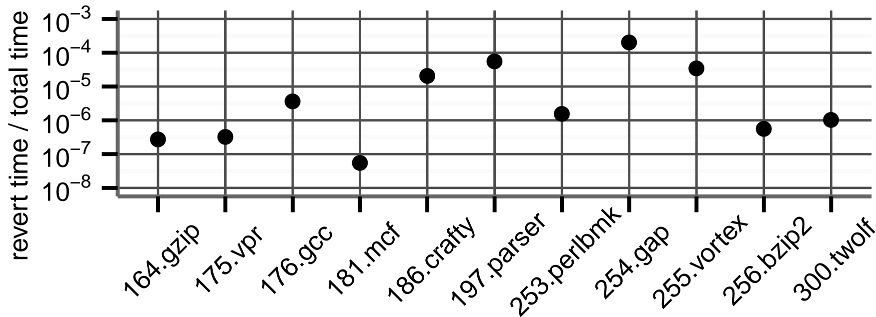
- Permutation history buffer tracks last 4 instructions
- Exception Stage: if a trap occurs, check permutation history buffer for `permi` instructions
- If any occur, go through history buffer in reverse order
  - For each `permi`: apply inverse permutation to permutation table



- `permi`s will be re-executed after trap handler

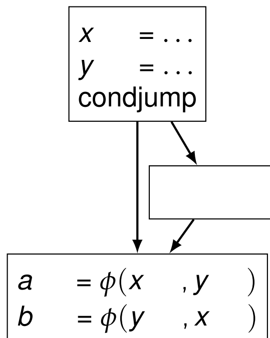
  ⇒ Register File in expected state

Reversion Effects



ϕ-functions

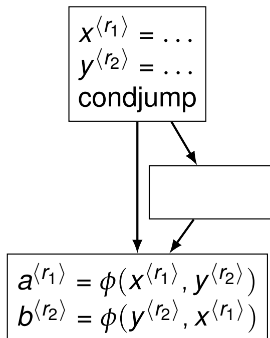

```
x = ...;
y = ...;
if (...) {
   t = x;
   x = y;
   y = t;
}
a = x;
b = y;
```



ϕ-functions

Karlsruhe Institute of Technology

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



ϕ-functions


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