Hardware Acceleration for Programs in SSA Form

Manuel Mohr, Artjom Grudnitsky, Tobias Modschiedler, Lars Bauer, Sebastian Hack, Jörg Henkel
SSA-Based Register Allocation

Front end

Parsing

Middle end

Optimizations

Back end

Register Allocation
SSA-Based Register Allocation

Not in Static Single Assignment Form

 Parsing

Optimizations

In Static Single Assignment Form

Register Allocation

Front end

Middle end

Back end
SSA-Based Register Allocation

Not in Static Single Assignment Form

In Static Single Assignment Form

Parsing

Optimizations

SSA-Based Register Allocation
SSA-Based Register Allocation

Not in Static Single Assignment Form

Front end

Middle end

Back end

In Static Single Assignment Form

Parsing

Fewer spills but more shuffle code

Optimizations

SSA-Based Register Allocation
Register Transfer Graphs

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

\[ r_1 \rightleftharpoons r_2 \rightarrow r_3 \quad r_4 \rightarrow r_5 \]

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

\[ r_1 \leftarrow r_2 \leftarrow r_3 \rightarrow r_4 \rightarrow r_5 \rightarrow r_6 \rightarrow r_7 \rightarrow r_8 \]

⇒ 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

```plaintext
r_1 ← r_2 ← r_3 ⇀ r_4 ⇀ r_5 ⇀ r_6 ⇀ r_7 → r_8
```

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

```plaintext
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

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

\[ r_1 \leftarrow r_2 \leftarrow r_3 \rightarrow r_4 \rightarrow r_5 \rightarrow r_6 \rightarrow r_7 \rightarrow r_8 \]

⇒ 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

\[ r_1 \leftrightarrow r_2 \leftrightarrow r_3 \leftrightarrow r_4 \leftrightarrow r_5 \leftrightarrow r_6 \rightarrow r_7 \rightarrow r_8 \]

⇒ 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

- Changing contents of multiple registers in one cycle very costly
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 $r_2$

![Diagram of register file showing $r_1$ pointing to 42 and $r_2$ pointing to 23 within the register file.](image)
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 $r_2$

$\Rightarrow$ Restriction to *permutations* of registers
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

\[
\begin{array}{cccccccc}
31 & 27 & 24 & 21 & 19 & 14 & 9 & 4 & 0 \\
0001 & a_1 & 000 & a_2 & b & c & d & e \\
\end{array}
\]
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

```
 31 27 24 21 19 14 9 4 0
0001 a1 000 a2 b c d e
```

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

\[ r_1 \overset{\text{perm}i5}{\rightarrow} r_2 \overset{\text{perm}i5}{\rightarrow} r_3 \overset{\text{perm}i5}{\rightarrow} r_4 \overset{\text{perm}i5}{\rightarrow} r_5 \]
Examples

\[ r_1 \leftrightarrow r_2 \leftrightarrow r_3 \leftrightarrow r_4 \rightarrow r_5 \]
\[ \text{permi5 } r_1, r_2, r_3, r_4, r_5 \]

\[ r_1 \leftrightarrow r_2 \]
\[ \text{permi5 } r_1, r_2 \]
Examples

\[ r_1 \leftrightarrow r_2 \leftrightarrow r_3 \rightarrow r_4 \rightarrow r_5 \quad \text{permi5 } r_1, r_2, r_3, r_4, r_5 \]

\[ r_1 \leftrightarrow r_2 \quad \text{permi5 } r_1, r_2 \]

\[ r_1 \leftrightarrow r_2 \leftrightarrow r_3 \leftrightarrow r_4 \quad \text{permi23 } r_1, r_2, r_3, r_4 \]
Goal: Generate efficient code using \texttt{perm}i instructions for all RTGs

Question: Which RTGs can be implemented using only \texttt{perm}i?
Goal: Generate efficient code using \texttt{perm} instructions for all RTGs

Question: Which RTGs can be implemented using only \texttt{perm}? 

RTGs in permutation form
- Permutation can be written as a product of cycles
- Cycles can be implemented with \texttt{perm} instructions
Goal: Generate efficient code using perm* instructions for all RTGs.

Question: Which RTGs can be implemented using only perm*?

RTGs in permutation form:
- Permutation can be written as a product of cycles.
- Cycles can be implemented with perm*s.

In general: RTGs can duplicate values:
- Permutations are injective.
- Value duplication impossible.
Two-Phase Approach

Arbitrary RTG

\[ \begin{align*}
  r_1 &\rightarrow r_3 \\
  r_3 &\rightarrow r_4 \\
  r_4 &\rightarrow r_7 \\
  r_5 &\rightarrow r_8 \\
  r_6 &\rightarrow r_9
\end{align*} \]
Two-Phase Approach

Arbitrary RTG

Phase 1:
Conversion

Phase 2:
Decomposition

\[\begin{align*}
&\text{mov } r3, r2 \\
&\text{mov } r6, r7 \\
&\text{mov } r4, r5 \\
\end{align*}\]
Two-Phase Approach

Arbitrary RTG

Phase 1:
Conversion

\[
\begin{align*}
    r_1 &\rightarrow r_3 \\
    r_3 &\rightarrow r_4 \\
    r_4 &\rightarrow r_7 \\
    r_5 &\rightarrow r_8 \\
    r_6 &\rightarrow r_9 \\
\end{align*}
\]

\[
\begin{align*}
    \text{mov } r_3, r_2 \\
    \text{mov } r_6, r_7 \\
    \text{mov } r_4, r_5
\end{align*}
\]

Phase 2:
Decomposition

\[
\begin{align*}
    \text{permi5 } r_1, r_3, r_4, r_6, r_9 \\
    \text{permi5 } r_5, r_8
\end{align*}
\]
Conversion into Permutation Form

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

\[
\begin{align*}
& \quad r_2 \\
& r_1 \rightarrow r_3 \rightarrow r_4 \rightarrow r_5 \rightarrow r_6
\end{align*}
\]
At each node with \( > 1 \) outgoing edge: keep edge that is part of longest path starting at node
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
At each node with $>1$ outgoing edge: keep edge that is part of longest path starting at node

\[
\begin{align*}
\text{r}_1 &\rightarrow \text{r}_3 \rightarrow \text{r}_4 \rightarrow \text{r}_5 \rightarrow \text{r}_6 \\
\text{r}_2 &\rightarrow \text{r}_3 \\
\end{align*}
\]
Conversion into Permutation Form

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

```
mov r3, r2
```
Decomposition into Cycles

- After conversion: Implement RTG in permutation form with as few permis as possible
- Need to combine multiple cycles to exploit permi23
Decomposition into Cycles

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

\[
\begin{align*}
& r_1 \iff r_2 \iff r_3 \\
& r_4 \iff r_5 \iff r_6 \iff r_7 \iff r_8 \iff r_9
\end{align*}
\]
Decomposition into Cycles

- After conversion: Implement RTG in permutation form with as few permis as possible
- Need to combine multiple cycles to exploit perm23

\[ \begin{align*}
& r_1 \rightarrow r_2 \rightarrow r_3 \\
& r_4 \rightarrow r_5 \rightarrow r_6 \rightarrow r_7 \rightarrow r_8 \rightarrow r_9 \\
& \text{perm}23
\end{align*} \]
Decomposition into Cycles

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

\[ r_1 \rightarrow r_2 \rightarrow r_3 \quad r_4 \rightarrow r_5 \rightarrow r_6 \rightarrow r_7 \rightarrow r_8 \rightarrow r_9 \]

permi23

permi5
Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

- Phase 1
  - While there is a cycle of size 4 or more: use $perm_5$ 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 \( \text{perm}_{5} \) to implement it

\[
\begin{align*}
r_1 & \quad r_2 \\
r_3 & \quad r_4
\end{align*}
\]

\( \text{perm}_{5} \)
Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

- Phase 1
  - While there is a cycle of size 4 or more: use $\text{perm}_5$ to implement it

\[
\begin{align*}
& r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow r_4 \rightarrow r_5 \rightarrow r_6 \rightarrow r_7 \\
& \text{perm}_5
\end{align*}
\]
Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

- **Phase 1**
  - While there is a cycle of size 4 or more: use \texttt{perm5} to implement it

- **Phase 2**: Only cycles of size $\leq 3$ left

- Phase 2: Only cycles of size $\leq 3$ left
Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

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

- **Phase 2**: Only cycles of size $\leq 3$ left
  - If 2-cycle and 3-cycle available: combine using \texttt{permi23}
  - If only 2-cycles available: combine in pairs using \texttt{permi23}
  - If only 3-cycles available: combine in groups of three using \texttt{permi23}
Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

- **Phase 1**
  - While there is a cycle of size 4 or more: use $\text{permi}_5$ to implement it

- **Phase 2**: Only cycles of size $\leq 3$ left
  - If 2-cycle and 3-cycle available: combine using $\text{permi}_{23}$
  - If only 2-cycles available: combine in pairs using $\text{permi}_{23}$

\[ r_1 \leftrightarrow r_2 \quad r_3 \leftrightarrow r_4 \quad r_5 \leftrightarrow r_6 \quad r_7 \leftrightarrow r_8 \]

$\text{permi}_{23}$ $\text{permi}_{23}$
Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

Phase 1
- While there is a cycle of size 4 or more: use perm5 to implement it

Phase 2: Only cycles of size ≤ 3 left
- If 2-cycle and 3-cycle available: combine using perm23
- If only 2-cycles available: combine in pairs using perm23
- If only 3-cycles available: combine in groups of three using perm23

\[ r_1 \leftrightarrow r_2 \leftrightarrow r_3 \quad r_4 \leftrightarrow r_5 \leftrightarrow r_6 \quad r_7 \leftrightarrow r_8 \leftrightarrow r_9 \]

\[ \text{perm23} \]

\[ \text{perm23} \]
Underlying architecture: Gaisler LEON3, 7-stage pipeline

Example: add r9 r5 r7

For permission support: modifications of **Decode** and **Exception** stages
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

![Diagram showing the process of fetching, decoding, and register operations with a permutation table.]
Applying new Permutations

- Applying permutation \texttt{permi5 r5 r9 r7 r6 r8}

Permutation applied in Decode stage \textit{(early committing)}

- No changes to forwarding logic required
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
<table>
<thead>
<tr>
<th>Backend (total)</th>
<th>Default [ms]</th>
<th>Our code gen. [ms]</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>63 598.0</td>
<td>63 927.0</td>
<td>+0.5%</td>
</tr>
</tbody>
</table>

- Code generation does not cause significant overhead
Four different register allocator configurations:

- **ILP**
- **Recoloring**
- **Biased**
- **Naive**

- Increasing RTG size
- Increasing number of RTGs
- Decreasing compilation time
## Code Quality

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

- Relative change of number of executed instructions

The worse the register allocation, the higher the benefit using our approach. Confirmation by FPGA measurements, speedup up to 1.07.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ILP</th>
<th>Recoloring</th>
<th>Biased</th>
<th>Naive</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>−0.7%</td>
<td>−1.0%</td>
<td>−1.9%</td>
<td>−16.4%</td>
</tr>
<tr>
<td>175.vpr</td>
<td>−0.3%</td>
<td>−0.3%</td>
<td>−1.0%</td>
<td>−3.4%</td>
</tr>
<tr>
<td>176.gcc</td>
<td>−0.4%</td>
<td>−0.5%</td>
<td>−2.7%</td>
<td>−11.4%</td>
</tr>
<tr>
<td>181.mcf</td>
<td>−1.9%</td>
<td>−1.9%</td>
<td>−2.8%</td>
<td>−7.8%</td>
</tr>
<tr>
<td>186.crafty</td>
<td>−1.0%</td>
<td>−0.8%</td>
<td>−3.9%</td>
<td>−15.2%</td>
</tr>
<tr>
<td>197.parser</td>
<td>−0.9%</td>
<td>−1.0%</td>
<td>−2.7%</td>
<td>−12.6%</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>−0.6%</td>
<td>−0.1%</td>
<td>−1.8%</td>
<td>−9.9%</td>
</tr>
<tr>
<td>254.gap</td>
<td>−0.3%</td>
<td>−0.9%</td>
<td>−2.0%</td>
<td>−7.1%</td>
</tr>
<tr>
<td>255.vortex</td>
<td>−0.5%</td>
<td>−0.8%</td>
<td>−5.1%</td>
<td>−15.1%</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>−0.3%</td>
<td>−0.6%</td>
<td>−3.1%</td>
<td>−11.3%</td>
</tr>
<tr>
<td>300.twolf</td>
<td>−0.3%</td>
<td>−0.3%</td>
<td>−0.8%</td>
<td>−1.9%</td>
</tr>
</tbody>
</table>

- 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 \textit{permis}
- Confirmation by FPGA measurements, speedup up to 1.07
# Area Overhead

<table>
<thead>
<tr>
<th></th>
<th>Base system</th>
<th>Our system</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>80 MHz</td>
<td>80 MHz</td>
<td>0%</td>
</tr>
<tr>
<td>BlockRAMs</td>
<td>28</td>
<td>28</td>
<td>0%</td>
</tr>
<tr>
<td>Flip-flops</td>
<td>7,607</td>
<td>8,851</td>
<td>16%</td>
</tr>
<tr>
<td>LUTs</td>
<td>15,024</td>
<td>21,630</td>
<td>44%</td>
</tr>
<tr>
<td>Slices</td>
<td>7,249</td>
<td>9,507</td>
<td>31%</td>
</tr>
</tbody>
</table>

- Frequency unaffected
- Logical-physical mapping ⇒ increase in FF usage
- Large multiplexers ⇒ increase in LUT usage
  - Considerably smaller overhead for ASIC implementation
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%
Backup Slides
\[ r_1 \rightarrow r_2 \rightarrow r_3 \quad \text{and} \quad r_4 \rightarrow r_5 \rightarrow r_6 \]
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:

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Register</th>
<th>Execute</th>
<th>Memory</th>
<th>Exception</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>permii</td>
<td>call</td>
<td>mov</td>
<td>-</td>
</tr>
</tbody>
</table>

mov
call
permi

mov

permi

executed twice – permutation applied twice → program crash

Instructions that commit after exception stage can be annulled

permi: revert effect of permutations executed before trap
## 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:

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Register</th>
<th>Execute</th>
<th>Memory permi</th>
<th>Exception call</th>
<th>Writeback mov</th>
</tr>
</thead>
</table>

Instructions that commit after exception stage can be annulled

`mov`  
`call`  
`permi`
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:

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Register</th>
<th>Execute</th>
<th>Memory</th>
<th>Exception</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>permi</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

mov
call
permi

→ Trap Handler

- permi executed twice – permutation applied twice → 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 \textit{permi} instructions
- If any occur, go through history buffer in reverse order
  - For each \textit{permi}: apply inverse permutation to permutation table

\begin{itemize}
  \item [\textbf{Decode}] \hspace{1cm} \textbf{Register} \hspace{1cm} \textbf{Execute} \hspace{1cm} \textbf{Memory} \hspace{1cm} \textbf{Exception}
  \item[\textbf{- - - - - - - -}] \hspace{1cm} \textbf{log phys} \hspace{1cm} \textbf{8 6 7 9 5} \hspace{1cm} \textbf{- - - - - - - -}
  \item[\textbf{select}] \hspace{1cm} \textbf{r5 r8} \hspace{1cm} \textbf{r6 r9} \hspace{1cm} \textbf{r7 r6} \hspace{1cm} \textbf{r8 r5} \hspace{1cm} \textbf{r9 r7}
  \item[\textbf{retrieve old permutation}] \hspace{1cm} \textbf{generate new permutation}
  \item[\textbf{r5 \rightarrow r5}] \hspace{1cm} \textbf{Permutation cycle: 5 9 7 6 8}
  \item[\textbf{r6 \rightarrow r9}] \hspace{1cm} \textbf{Permutation cycle: - - - - - - - -}
  \item[\textbf{r7 \rightarrow r7}] \hspace{1cm} \textbf{Permutation cycle: - - - - - - - -}
  \item[\textbf{r8 \rightarrow r8}] \hspace{1cm} \textbf{Permutation cycle: - - - - - - - -}
\end{itemize}

\textbullet \ \textit{permis} will be re-executed after trap handler
⇒ Register File in expected state
Reversion Effects

![Graph showing revert time vs. total time for various benchmarks]

- Benchmarks: 164.gzip, 175.vpr, 176.gcc, 181.mcf, 186.crafty, 197.parser, 253.perlbmk, 254.gap, 255.vortex, 256.bzip2, 300.twolf

- X-axis: Benchmarks
- Y-axis: Revert time / total time (in log scale)

The graph illustrates the revert time relative to the total time for each benchmark.
\( \phi \)-functions

\[
\begin{align*}
x &= \ldots; \\
y &= \ldots; \\
\text{if } ( \ldots ) &\{ \\
&\quad t = x; \\
&\quad x = y; \\
&\quad y = t; \\
\} \\
a &= x; \\
b &= y;
\end{align*}
\]

\[
\begin{align*}
x &= \ldots \\
y &= \ldots \\
\text{condjump} \\
a &= \phi(x, y) \\
b &= \phi(y, x)
\end{align*}
\]
\( \phi \)-functions

\[
\begin{align*}
x & = \ldots ; \\
y & = \ldots ; \\
\textbf{if} \ (\ldots) \ {\{} \\
\quad & t = x ; \\
\quad & x = y ; \\
\quad & y = t ; \\
\} \\
\textit{a} & = x ; \\
\textit{b} & = y ;
\end{align*}
\]

\[
\begin{align*}
\textit{x}^{\langle r_1 \rangle} & = \ldots \\
\textit{y}^{\langle r_2 \rangle} & = \ldots \\
\textbf{condjump} \\
\textit{a}^{\langle r_1 \rangle} & = \phi(\textit{x}^{\langle r_1 \rangle}, \textit{y}^{\langle r_2 \rangle}) \\
\textit{b}^{\langle r_2 \rangle} & = \phi(\textit{y}^{\langle r_2 \rangle}, \textit{x}^{\langle r_1 \rangle})
\end{align*}
\]
$\phi$-functions

\[
x = \ldots;
y = \ldots;
\text{if } (\ldots) \{ 
  t = x;
x = y;
y = t;
\}
\]
\[
a = x;
b = y;
\]

\[
x^{r_1} = \ldots
y^{r_2} = \ldots
\]

\[
\text{condjump}
\]

\[
a^{r_1} = \phi(x^{r_1}, y^{r_1})
b^{r_2} = \phi(y^{r_2}, x^{r_2})
\]

\[r_1 \leftrightarrow r_2\]