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 destroyed before register allocation.
- Recent research: SSA-based register allocation.

Shuffle Code

- g-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 V9 ISA:
- 32 integer registers ⇒ 5 bits to identify one register.
- 7 bits for opcode ⇒ 25 bits left for encoding 5 register numbers.

Two new instructions:
- perm15: Apply one cyclic permutation with up to 5 elements.
- perm23: Apply two independent cycles with 2 and up to 3 elements.

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 perm1 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 \( \geq 1 \) outgoing edge:
  - Keep edge that is part of longest path starting at node.

Output: RTG in permutation form ⇒ list of copy instructions.

Phase 2: Decomposition into Cycles

Input: RTG in permutation form

Output: List of perm1 instructions that implement RTG.

- Greedy algorithm with linear runtime shown on right.

Hardware Implementation

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 perm1 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 \( \geq 1 \) outgoing edge:
  - Keep edge that is part of longest path starting at node.

Output: RTG in permutation form ⇒ list of copy instructions.

Phase 2: Decomposition into Cycles

Input: RTG in permutation form

Output: List of perm1 instructions that implement RTG.

- Greedy algorithm with linear runtime shown on right.

Hardware Implementation

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 perm1 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 \( \geq 1 \) outgoing edge:
  - Keep edge that is part of longest path starting at node.

Output: RTG in permutation form ⇒ list of copy instructions.

Phase 2: Decomposition into Cycles

Input: RTG in permutation form

Output: List of perm1 instructions that implement RTG.

- Greedy algorithm with linear runtime shown on right.

Hardware Implementation

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 perm1 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 \( \geq 1 \) outgoing edge:
  - Keep edge that is part of longest path starting at node.

Output: RTG in permutation form ⇒ list of copy instructions.

Phase 2: Decomposition into Cycles

Input: RTG in permutation form

Output: List of perm1 instructions that implement RTG.

- Greedy algorithm with linear runtime shown on right.

Hardware Implementation

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 perm1 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 \( \geq 1 \) outgoing edge:
  - Keep edge that is part of longest path starting at node.

Output: RTG in permutation form ⇒ list of copy instructions.

Phase 2: Decomposition into Cycles

Input: RTG in permutation form

Output: List of perm1 instructions that implement RTG.

- Greedy algorithm with linear runtime shown on right.