Optimal Shuffle Code with Permutation Instructions

Sebastian Buchwald, Manuel Mohr, and Ignaz Rutter

Karlsruhe Institute of Technology
{sebastian.buchwald, manuel.mohr, rutter}@kit.edu

Abstract. During compilation of a program, register allocation is the task of mapping program variables to machine registers. During register allocation, the compiler may introduce *shuffle code*, consisting of copy and swap operations, that transfers data between the registers. Three common sources of shuffle code are conflicting register mappings at joins in the control flow of the program, e.g, due to if-statements or loops; the calling convention for procedures, which often dictates that input arguments or results must be placed in certain registers; and machine instructions that only allow a subset of registers to occur as operands.
Recently, Mohr et al. [9] proposed to speed up shuffle code with special hardware instructions that arbitrarily permute the contents of up to five registers and gave a heuristic for computing such shuffle codes.
In this paper, we give an efficient algorithm for generating optimal shuffle code in the setting of Mohr et al. An interesting special case occurs when no register has to be transferred to more than one destination, i.e., it suffices to permute the contents of the registers. This case is equivalent to factoring a permutation into a minimal product of permutations, each of which permutes up to five elements.

1 Introduction

One of the most important tasks of a compiler during code generation is register allocation, which is the task of mapping program variables to machine registers. During this phase, it is frequently necessary to insert so-called *shuffle code* that transfers values between registers. Common reasons for the insertion of shuffle code are control flow joins, procedure calling conventions and constrained machine instructions.

The specification of a shuffle code, i.e., a description which register contents should be transferred to which registers, can be formulated as a directed graph whose vertices are the registers and an edge (u, v) means that the content of u before the execution of the shuffle code must be in v after the execution. Naturally, every vertex must have at most one incoming edge. Note that vertices may have several outgoing edges, indicating that their contents must be transferred to several destinations, and even loops (u, u), indicating that the content of register u must be preserved. We call such a graph a *Register Transfer Graph* or *RTG*. Two important special types of RTGs are outdegree-1 RTGs where the maximum out-degree is 1 and PRTGs where $\deg^-(v) = \deg^+(v) = 1$ for all vertices v ($\deg^-$ and $\deg^+$ denote the in- and out-degree of a vertex, respectively).

We say that a shuffle code, consisting of a sequence of copy and swap operations on the registers, *implements* an RTG if after the execution of the shuffle code every register whose corresponding vertex has an incoming edge has the correct content. The *shuffle code generation* problem asks for a shortest shuffle code that implements a given RTG.

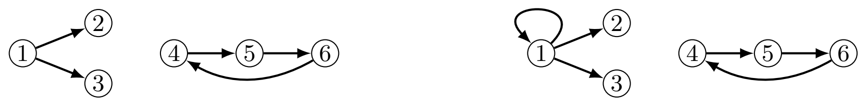


Fig. 1: Two example RTGs where the optimal shuffle code is not obvious.

The amount of shuffle code directly depends on the quality of copy coalescing, a subtask of register allocation [9]. As copy coalescing is NP-complete [2], reducing the amount of shuffle code is expensive in terms of compilation time, and thus cannot be afforded in all contexts, e.g., just-in-time compilation.

Therefore, it has been suggested to allow more complicated operations than simply copying and swapping to enable more efficient shuffle code. Mohr et al. [9] propose to allow performing permutations on the contents of small sets of up to five registers. The processor they develop offers three instructions to implement shuffle code:

`copy`**:** copies the content of one register to another one
`permi5`**:** cyclically shifts the contents of up to five registers
`permi23`**:** swaps the contents of two registers and performs a cyclic shift of the contents of up to three registers; the two sets of registers must be disjoint.

In fact, the two operations `permi5` and `permi23` together allow to arbitrarily permute the contents of up to five registers in a single operation. A corresponding hardware and a modified compiler that employs a greedy approach to generate the shuffle code have been shown to improve performance in practice [9]. While the greedy heuristic works well in practice, it does not find an optimal shuffle code in all cases.

It is not obvious how to generate optimal shuffle code using the three instructions `copy`, `permi5` and `permi23` even for small RTGs. In the left RTG from Fig. 1, a naive solution would implement edges $(1, 2)$ and $(1, 3)$ using copies and the remaining cycle $(4\ 5\ 6)$ using a `permi5`. However, using one `permi23` to implement the cycle $(4\ 5\ 6)$ and swap registers 1 and 2, and then copying register 2 to 3 requires only two instructions. This is legal because the contents of register 1 can be overwritten. The same trick is not applicable for the right RTG in Fig. 1 because of the loop $(1, 1)$ and hence three instructions are necessary to implement that RTG.

A maximum permutation size of 5 may seem arbitrary at first but is a consequence of instruction encoding constraints. In each `permi` instruction, the register numbers and their order must be encoded in the instruction word. Hence, $\lceil \log_2 \left(\binom{n}{k} k!\right) \rceil$ bits of an instruction word are needed to be able to encode all permutations of k registers out of n total registers. As many machine architectures use a fixed size for instruction words, e.g., 32 or 64 bits, and the operation type must also be encoded in the instruction word, space is very limited. In fact, for a 32 bit instruction word, 34 is the maximum number of registers that leave enough space for the operation type.

Related Work. As long as only copy and swap operations are allowed, finding an optimal shuffle code for a given RTG is a straightforward task [7, p. 56–57]. Therefore work in the area of compiler construction in this context has focused on coalescing techniques that reduce the number and the size of RTGs [1, 2, 6, 8].

From a theoretical point of view, the most closely related work studies the case where the input RTG consists of a union of disjoint directed cycles, which can be interpreted as a permutation π. Then, no copy operations are necessary for an optimal shuffle code and hence the problem of finding an optimal shuffle code using `permi23` and `permi5` is equivalent to writing π as a shortest product of permutations of maximum size 5, where a permutation of n elements has size k if it fixes $n - k$ elements.

There has been work on writing a permutation as a product of permutations that satisfy certain restrictions. The factorization problem on permutation groups from computational group theory [10] is the task of writing an element g of a permutation group as a product of given generators S. Hence, an algorithm for solving the factorization problem could be applied in our context by using all possible permutations of size 5 or less as the set S. However, the algorithms do not guarantee minimality of the product. For the case that S consists of all permutations that reverse a contiguous subsequence of the elements, known as the pancake sorting problem, it has been shown that computing a factoring of minimum size is NP-complete [4].

Farnoud and Milenkovic [5] consider a weighted version of factoring a permutation into transpositions. They present a polynomial constant-factor approximation algorithm for factoring a given permutation into transpositions where transpositions have arbitrary non-negative costs. In our problem, we cannot assign costs to an individual transposition as its cost is context-dependent, e.g., four transpositions whose product is a cycle require one operation, whereas four arbitrary transpositions may require two.

Contribution and Outline. In this paper, we present an efficient algorithm for generating optimal shuffle code using the operations `copy`, `permi5`, and `permi23`, or equivalently, using copy operations and permutations of size at most 5.

We first prove the existence of a special type of optimal shuffle codes whose copy operations correspond to edges of the input RTG in Section 2. Removing the set of edges implemented by copy operations from an RTG leaves an outdegree-1 RTG.

We show that the greedy algorithm proposed by Mohr et al. [9] finds optimal shuffle codes for outdegree-1 RTGs and that the size of an optimal shuffle code can be expressed as a function that depends only on three characteristic numbers of the outdegree-1 RTG rather than on its structure. Since PRTGs are a special case of outdegree-1 RTGs, this shows that GREEDY is a linear-time algorithm for factoring an arbitrary permutation into a minimum number of permutations of size at most 5.

Finally, in Section 4, we show how to compute an optimal set of RTG edges that will be implemented by copy operations such that the remaining outdegree-1 RTG admits a shortest shuffle code. This is done by several dynamic programs for the cases that the input RTG is disconnected, is a tree, or is connected and contains a (single) cycle. Proofs omitted due to space constraints can be found in the full version of this paper [3].

2 Register Transfer Graphs and Optimal Shuffle Codes

In this section, we rephrase the shuffle code generation problem as a graph problem. An RTG that has only self-loops needs no shuffle-code and is called *trivial*.

It is easy to define the effect of a permutation on an RTG. Let G be an RTG and let π be an arbitrary permutation that is applied to the contents of the registers. We define $\pi G = (V, \pi E)$, where $\pi E = \{(\pi(u), v) \mid (u, v) \in E\}$. This models the fact that if v should receive the data contained in u, then after π moves the data contained in u to some other register $\pi(u)$, the data contained in $\pi(u)$ should end up in v. We observe that for two permutations π_1, π_2 of V, it is $(\pi_2 \circ \pi_1)G = \pi_2(\pi_1(G))$, i.e., we have defined a group action of the symmetric group on RTGs. For PRTGs, the shuffle code generation problem asks for a shortest shuffle code that makes the given PRTG trivial.

Unfortunately, it is not possible to directly express copy operations in RTGs. Instead, we rely on the following observation. Consider an arbitrary shuffle code that contains a copy $a \to b$ with source a and target b that is followed by a transposition τ of the contents of registers c and d. We can replace this sequence by a transposition of the registers $\{c, d\}$ and a copy $\tau(a) \to \tau(b)$. Thus, given a sequence of operations, we can successively move the copy operations to the end of the sequence without increasing its length. Thus, for any RTG there exists a shuffle code that consists of a pair of sequences $((\pi_1, \dots, \pi_p), (c_1, \dots, c_t))$, where the π_i are permutation operations and the c_i are copy operations. We now strengthen our assumption on the copy operations.

Lemma 1. *Every instance of the shuffle code generation problem has an optimal shuffle code* $((\pi_1, \dots, \pi_p), (c_1, \dots, c_t))$ *such that*

(i) *No register occurs as both a source and a target of copy operations.*
(ii) *Every register is the target of at most one copy operation.*
(iii) *There is a bijection between the copy operations* c_i *and the edges of* πG *that are not loops, where* $\pi = \pi_p \circ \pi_{p-1} \circ \cdots \circ \pi_1$.
(iv) *If* u *is the source of a copy operation, then* u *is incident to a loop in* πG.
(v) *The number of copies is* $\sum_{v \in V} \max\{\deg^+_G(v) - 1, 0\}$.

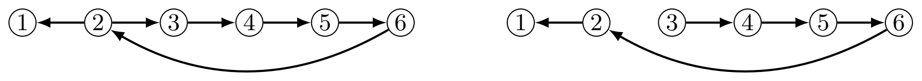
We call a shuffle code satisfying the conditions of Lemma 1 *normalized*. Observe that the number of copy operations used by a normalized shuffle code is a lower bound on the number of necessary copy operations since permutations cannot copy values.

Consider now an RTG G together with a normalized optimal shuffle code and one of its copy operations $u \to v$. Since the code is normalized, the value transferred to v by this copy operation is the one that stays there after the shuffle code has been executed. If v had no incoming edge in G, then we could shorten the shuffle by omitting the copy operation. Thus, v has an incoming edge (u', v) in G, and we associate the copy $u \to v$ with the edge (u', v) of G. In fact, $u' = \pi^{-1}(u)$, where $\pi = \pi_p \circ \cdots \circ \pi_1$. In this way, we associate every copy operation with an edge of the input RTG. In fact, this is an injective mapping by Lemma 1 (ii).

Lemma 2. *Let* $((\pi_1, \dots, \pi_p), (c_1, \dots, c_t))$ *be an optimal shuffle code* S *for an RTG* $G = (V, E)$ *and let* $C \subseteq E$ *be the edges that are associated with copies in* S. *Then*

(i) *Every vertex* v *has* $\max\{\deg^+_G(v) - 1, 0\}$ *outgoing edges in* C.
(ii) $G - C$ *is an outdegree-1 RTG.*
(iii) $\pi_1, \dots, \pi_p$ *is an optimal shuffle code for* $G - C$.

Lemma 2 shows that an optimal shuffle code for an RTG G can be found by 1) picking for each vertex one of its outgoing edges (if it has any) and removing the remaining



(a) The original RTG G needs one permutation and one copy operation.

(b) After removing the edge $(2, 3)$, the RTG needs two permutation operations.

Fig. 2: The RTG G obtains the normalized optimal shuffle code (π_1, c_1), where $\pi_1 = (23456)$ and $c_1 = 3 \to 1$. However, after removing the edge $(2, 3)$ (instead of $(1, 2)$) we cannot achieve an optimal solution anymore.

edges from G, 2) finding an optimal shuffle code for the resulting outdegree-1 RTG, and 3) creating one copy operation for each of the previously removed edges. Fig. 2 shows that the choice of the outgoing edges is crucial to obtain an optimal shuffle code.

In the following, we first show how to compute an optimal shuffle code for an outdegree-1 RTG in Section 3. Afterwards, in Section 4, we design an algorithm for efficiently determining a set of edges to be removed such that the resulting outdegree-1 RTG admits a shuffle code with the smallest number of operations.

3 Optimal Shuffle Code for Outdegree-1 RTGs

In this section we prove the optimality of the greedy algorithm proposed by Mohr et al. [9] for outdegree-1 RTGs. Before we formulate the algorithm, let us look at the effect of applying a transposition $\tau = (u\ v)$ to contiguous vertices of a k-cycle $K = (V_K, E_K)$ in a PRTG G, where k-cycle denotes a cycle of size k. Hence, $u, v \in V_K$ and $(u, v) \in E_K$. Then, in τG, the cycle K is replaced by a $(k-1)$-cycle and a vertex v with a loop. We say that τ has reduced the size of K by 1. If τK is trivial, we say that τ resolves K. It is easy to see that `permi5` reduces the size of a cycle by up to 4 and `permi23` reduces the sizes of two distinct cycles by 1 and up to 2, respectively. We can now formulate GREEDY as follows.

1. Complete each directed path of the input outdegree-1 RTG into a directed cycle, thereby turning the input into a PRTG.
2. While there exists a cycle K of size at least 4, apply a `permi5` operation to reduce the size of K as much as possible.
3. While there exist a 2-cycle and a 3-cycle, resolve them with a `permi23` operation.
4. Resolve pairs of 2-cycles by `permi23` operations.
5. Resolve triples of 3-cycles by pairs of `permi23` operations.

We claim that GREEDY computes an optimal shuffle code. Let G be an outdegree-1 RTG and let Q denote the set of paths and cycles of G. For a path or cycle $\sigma \in Q$, we denote by $\operatorname{size}(\sigma)$ the number of vertices of σ. Define $X = \sum_{\sigma \in Q} \lfloor \operatorname{size}(\sigma)/4 \rfloor$ and $a_i = |\{\sigma \in Q \mid \operatorname{size}(\sigma) = i \mod 4\}|$ for $i = 2, 3$. We call the triple $\operatorname{sig}(G) = (X, a_2, a_3)$ the *signature* of G.

Lemma 3. *Let G be an outdegree-1 RTG with* $\operatorname{sig}(G) = (X, a_2, a_3)$. *The number* GREEDY$(G)$ *of operations in the shuffle code produced by the greedy algorithm is* GREEDY$(G) = X + \max\{\lceil (a_2 + a_3)/2 \rceil, \lceil (a_2 + 2a_3)/3 \rceil\}$.

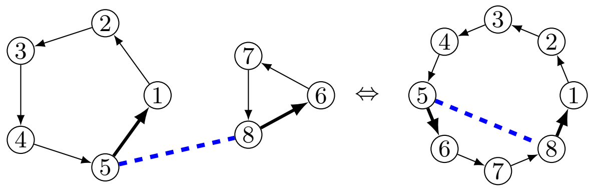


Fig. 3: The transposition $\tau = (5\ 8)$ acting on PRTGs. Affected edges are drawn thick. Read from left to right, the transposition is a merge; read from right to left, it is a split.

In particular, the length of the shuffle code computed by GREEDY only depends on the signature of the input RTG G. In the following, we prove that GREEDY is optimal for outdegree-1 RTGs and hence GREEDY(G) is the length of an optimal shuffle code.

Lemma 4. *Let G, G' be PRTGs with $\operatorname{sig}(G) = (X, a_2, a_3)$, $\operatorname{sig}(G') = (X', a_2', a_3')$ and* GREEDY(G) − GREEDY$(G') \geq c$*, and let $(\Delta_X, \Delta_2, \Delta_3) = \operatorname{sig}(G) - \operatorname{sig}(G')$. If $a_2 \geq a_3$, then $2\Delta_X + \Delta_2 + \Delta_3 \leq -2c+1$. If $a_3 > a_2$, then $3\Delta_X + \Delta_2 + 2\Delta_3 \leq -3c+2$.*

Proof (sketch). We assume that $a_2 \geq a_3$, the other case is analogous. By Lemma 3 GREEDY$(G) \leq X + (a_2 + a_3 + 1)/2$ and GREEDY$(G') \geq X' + (a_2' + a_3')/2$. Therefore, GREEDY$(G)$ − GREEDY$(G') \leq -\Delta_X - (\Delta_2 + \Delta_3 - 1)/2 = -(2\Delta_X + \Delta_2 + \Delta_3 - 1)/2$. By assumption, $-(2\Delta_X + \Delta_2 + \Delta_3 - 1)/2 \geq c$; this is equivalent to the claim. □

Lemma 4 gives us necessary conditions for when the GREEDY solutions of two RTGs differ by some value c. These necessary conditions depend only on the difference of the two signatures. To study them more precisely, we define $\Psi_1(\Delta_X, \Delta_2, \Delta_3) = 2\Delta_X + \Delta_2 + \Delta_3$ and $\Psi_2(\Delta_X, \Delta_2, \Delta_3) = 3\Delta_X + \Delta_2 + 2\Delta_3$. Next, we study the effect of a single transposition on these two functions.

Let $G = (V, E)$ be a PRTG with $\operatorname{sig}(G) = (X, a_2, a_3)$ and let τ be a transposition of two elements in V. We distinguish cases based on whether the swapped elements are in different connected components or not. In the former case, we say that τ is a *merge*, in the latter we call it a *split*; see Fig. 3 for an illustration.

We start with the merge operations. When merging two cycles of size s_1 and s_2, respectively, they are replaced by a single cycle of size $s_1 + s_2$. Note that removing the two cycles may decrease the values a_2 and a_3 of the signature by at most 2 in total. The new cycle can potentially increase one of these values by 1. The value X never decreases, and it increases by 1 if and only if $s_1 \mod 4 + s_2 \mod 4 \geq 4$. Table 1a shows the possible signature changes $(\Delta_X, \Delta_2, \Delta_3)$ resulting from a merge. The entry in row i and column j shows the result of merging two cycles whose sizes modulo 4 are i and j, respectively. Table 1b shows the corresponding values of Ψ_1 and Ψ_2. Only entries with $i \leq j$ are shown, the remaining cases are symmetric.

Lemma 5. *Let G be a PRTG with $\operatorname{sig}(G) = (X, a_2, a_3)$ and let τ be a merge. Then* GREEDY$(G) \leq$ GREEDY(τG).

Proof. Suppose GREEDY$(\tau G) <$ GREEDY(G). Then GREEDY(G) − GREEDY$(\tau G) \geq 1$ and by Lemma 4 either $\Psi_1 \leq -1$ or $\Psi_2 \leq -1$. However, Table 1b shows the values of Ψ_1 and Ψ_2 for all possible merges. In all cases it is $\Psi_1, \Psi_2 \geq 0$. A contradiction. □

	0	1	2	3
0	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$
1		$(0,1,0)$	$(0,-1,1)$	$(1,0,-1)$
2			$(1,-2,0)$	$(1,-1,-1)$
3				$(1,1,-2)$

(a) Signature change $(\mathit{\Delta}_X, \mathit{\Delta}_2, \mathit{\Delta}_3)$.

	0	1	2	3
0	0	0	0	0
1		1	0	1
2			0	0
3				1

	0	1	2	3
0	0	0	0	0
1		1	1	1
2			1	0
3				0

(b) Values of $\mathit{\Psi}_1$ (left) and $\mathit{\Psi}_2$ (right).

Table 1: Signature changes and $\mathit{\Psi}$ values for merges. Row and column are the cycle sizes modulo 4 before the merge.

In particular, the lemma shows that merges never decrease the cost of the greedy solution, even if they were for free. We now make a similar analysis for splits. It is, however, obvious that splits indeed may decrease the cost of greedy solutions. In fact, one can always split cycles in a PRTG until it is trivial.

First, we study again the effect of splits on the signature change $(\mathit{\Delta}_X, \mathit{\Delta}_2, \mathit{\Delta}_3)$. Since a split is an inverse of a merge, we can essentially reuse Table 1a. If merging two cycles whose sizes modulo 4 are i and j, respectively, results in a signature change of $(\mathit{\Delta}_X, \mathit{\Delta}_2, \mathit{\Delta}_3)$, then, conversely, we can split a cycle whose size modulo 4 is $i + j$ into two cycles whose sizes modulo 4 are i and j, respectively, such that the signature change is $(-\mathit{\Delta}_X, -\mathit{\Delta}_2, -\mathit{\Delta}_3)$, and vice versa. Note that given a cycle whose size modulo 4 is s one has to look at all cells (i, j) with $i + j \equiv s \pmod 4$ to consider all the possible signature changes. Since $\mathit{\Psi}_1, \mathit{\Psi}_2$ are linear, negating the signature change also negates the corresponding value. Thus, we can reuse Table 1b for splits by negating each entry.

Lemma 6. *Let $G = (V, E)$ be a PRTG and let π be a cyclic shift of c vertices in V. Let further $(\mathit{\Delta}_X, \mathit{\Delta}_2, \mathit{\Delta}_3)$ be the signature change affected by π. Then $\mathit{\Psi}_1(\mathit{\Delta}_X, \mathit{\Delta}_2, \mathit{\Delta}_3) \geq -\lceil (c-1)/2 \rceil$ and $\mathit{\Psi}_2(\mathit{\Delta}_X, \mathit{\Delta}_2, \mathit{\Delta}_3) \geq -\lceil (3c-3)/4 \rceil$.*

Proof. We can write $\pi = \tau_{c-1} \circ \cdots \circ \tau_1$ as a product of $c-1$ transpositions such that any two consecutive transpositions τ_i and τ_{i+1} affect a common element for $i = 1, \ldots, c-1$.

Each transposition decreases $\mathit{\Psi}_1$ (or $\mathit{\Psi}_2$) by at most 1, but a decrease happens only for certain split operations. However, it is not possible to reduce $\mathit{\Psi}_1$ (or $\mathit{\Psi}_2$) with every single transposition since for two consecutive splits the second has to split one of the connected components resulting from the previous splits. To get an overview of the sequences of splits that reduce the value of $\mathit{\Psi}_1$ (or of $\mathit{\Psi}_2$) by 1 for each split, we consider the following transition graphs T_k for $\mathit{\Psi}_k$ ($k = 1, 2$) on the vertex set $S = \{0, 1, 2, 3\}$. In the graph T_k there is an edge from i to j if there is a split that splits a component of size $i \mod 4$ such that one of the resulting components has size $j \mod 4$ and this split decreases $\mathit{\Psi}_k$ by 1. The transition graphs T_1 and T_2 are shown in Fig. 4.

For $\mathit{\Psi}_1$ the longest path in the transition graph has length 1. Thus, the value of $\mathit{\Psi}_1$ can be reduced at most every second transposition and $\mathit{\Psi}_1(\mathit{\Delta}_X, \mathit{\Delta}_2, \mathit{\Delta}_3) \geq -\lceil (c-1)/2 \rceil$.

For $\mathit{\Psi}_2$ the longest path has length 3 (vertex 1 has out-degree 0). Therefore, after at most three consecutive steps that decrease $\mathit{\Psi}_2$, there is one that does not. It follows that at least $\lfloor (c-1)/4 \rfloor$ operations do not decrease $\mathit{\Psi}_2$, and consequently at most $\lceil (3c-3)/4 \rceil$ operations decrease $\mathit{\Psi}_2$ by 1. Thus, $\mathit{\Psi}_2(\mathit{\Delta}_X, \mathit{\Delta}_2, \mathit{\Delta}_3) \geq -\lceil (3c-3)/4 \rceil$. □

$$\begin{tikzcd}
0 \arrow[r] \arrow[d] & 1 \\
3 & 2 \arrow[l] \arrow[u]
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
0 \arrow[r] \arrow[d] \arrow[dr] & 1 \\
3 \arrow[r] \arrow[ur] & 2 \arrow[u]
\end{tikzcd}$$

Fig. 4: Transition graphs for Ψ_1 (left) and Ψ_2 (right).

Since `permi5` performs a single cyclic shift and `permi23` is the concatenation of two cyclic shifts, Lemmas 6 and 4 can be used to show that no such operation may decrease the number of operations GREEDY has to perform by more than 1.

Corollary 1. *Let G be a PRTG and let π be an operation, i.e., either a* `permi23` *or a* `permi5`. *Then* $\text{GREEDY}(G) \leq \text{GREEDY}(\pi G) + 1$.

Using this corollary and an induction on the length of an optimal shuffle code, we show that GREEDY is optimal for PRTGs; if no operation reduces the number of operations GREEDY needs by more than 1, why not use the operation suggested by GREEDY?

Theorem 1. *Let G be a PRTG. An optimal shuffle code for G takes* $\text{GREEDY}(G)$ *operations. Algorithm* GREEDY *computes an optimal shuffle code in linear time.*

Moreover, since merge operations may not decrease the cost of GREEDY and any PRTG that can be formed from the original outdegree-1 RTG G by inserting edges can be obtained from the PRTG G' formed by GREEDY and a sequence of merge operations, it follows that the length of an optimal shuffle for G is $\text{GREEDY}(G')$. Thus, GREEDY is optimal for outdegree-1 RTGs.

Theorem 2. *Let G be an outdegree-1 RTG. Then an optimal shuffle code for G requires* $\text{GREEDY}(G)$ *operations.* GREEDY *computes such a shuffle code in linear time.*

4 The General Case

In this section we study the general case. A *copy set* of an RTG $G = (V, E)$ is a set $C \subseteq E$ such that $G - C = (V, E - C)$ is an outdegree-1 RTG and $|C| = \sum_{v \in V} \max\{\deg^+(v) - 1, 0\}$. We denote by $\mathcal{C}(G)$ the set of all copy sets of G. According to Lemma 2 an optimal shuffle code for G can be found by finding a copy set $C \in \mathcal{C}(G)$ such that the outdegree-1 RTG $G - C$ admits a shortest shuffle code. By Theorem 2 an optimal shuffle code for $G - C$ can be computed with the greedy algorithm and its length can be computed according to Lemma 3. We thus seek a copy set $C \in \mathcal{C}(G)$ that minimizes the cost function $\text{GREEDY}(G - C) = X + \max\{\lceil (a_2 + a_3)/2 \rceil, \lceil (a_2 + 2a_3)/3 \rceil\}$, where (X, a_2, a_3) is the signature of $G - C$. Such a copy set is called *optimal*. Clearly, this is equivalent to minimizing the function

$$\text{GREEDY}'(G - C) = X + \max\{\frac{a_2 + a_3}{2}, \frac{a_2 + 2a_3}{3}\} = \begin{cases} X + \frac{a_2}{2} + \frac{a_3}{2} & \text{if } a_2 \geq a_3 \\ X + \frac{a_2}{3} + \frac{2a_3}{3} & \text{if } a_2 < a_3 \end{cases}$$

To keep track of which case is used for evaluating GREEDY$'$, we define $\mathrm{diff}(G-C) = a_2 - a_3$ and compute for each of the two function parts and every possible value d a copy set C_d with $\mathrm{diff}(G - C_d) = d$ that minimizes that function. More formally, we define $\mathrm{cost}^1(G-C) = X + \frac{1}{2}a_2 + \frac{1}{2}a_3$ and $\mathrm{cost}^2(G-C) = X + \frac{1}{3}a_2 + \frac{2}{3}a_3$ and we seek two tables $T^1_G[\cdot], T^2_G[\cdot]$, such that $T^i_G[d]$ is the smallest cost $\mathrm{cost}^i(G-C)$ that can be achieved with a copy set $C \in \mathcal{C}(G)$ with $\mathrm{diff}(G-C) = d$. We observe that $T^i_G[d] = \infty$ for $d < -n$ and for $d > n$. The following lemma shows that the length of an optimal shuffle code can be computed from these two tables.

Lemma 7. *Let $G = (V, E)$ be an RTG. The length of an optimal shuffle code for G is $\sum_{v\in V} \max\{\deg^+(v) - 1, 0\} + \min\{\min_{d\geq 0}\lceil T^1_G[d]\rceil, \min_{d<0}\lceil T^2_G[d]\rceil\}$.*

In the following, we show how to compute for an RTG G a table $T_G[\cdot]$ with

$$T_G[d] = \min_{\substack{C\in\mathcal{C}(G)\\ \mathrm{diff}(G-C)=d}} \mathrm{cost}(G-C)$$

for an arbitrary cost function $\mathrm{cost}(G-C) = c(\mathrm{sig}(G-C))$, where c is a linear function. This is done in several steps depending on whether G is disconnected, is a tree, or is connected and contains a cycle. Before we continue, we introduce several preliminaries to simplify the following calculations. We denote by P_s a directed path on s vertices.

Definition 1. *A map f that assigns a value to an outdegree-1 RTG is* signature-linear *if there exists a linear function $g\colon \mathbb{R}^3 \to \mathbb{R}$ such that $f(G) = g(\mathrm{sig}(G))$ for every outdegree-1 RTG G. For a signature-linear function f, $\Delta_f(s) = f(P_{s+1}) - f(P_s)$ is the* correction term.

Note that both $\mathrm{cost} = c \circ \mathrm{sig}$ and $\mathrm{diff} = d \circ \mathrm{sig}$ with $d(X, a_2, a_3) = a_2 - a_3$ are signature-linear. The correction term $\Delta_f(s)$ describes the change of f when the size of one connected component is increased from s to $s+1$.

Lemma 8. *Let f be a signature-linear function. Then the following hold:*
(i) $f(G_1 \cup G_2) = f(G_1) + f(G_2)$ for disjoint outdegree-1 RTGs G_1, G_2,
(ii) Let $G = (V, E)$ be an outdegree-1 RTG and let $v \in V$ with in-degree 0. Denote by s the size of the connected component containing v and let $G^+ = (V \cup \{u\}, E \cup \{(u, v)\})$ where u is a new vertex. Then $f(G^+) = f(G) + \Delta_f(s)$.

Note that $\Delta_f(s) = \Delta_f(s+4)$ for all values of s and hence it suffices to know the size of the enlarged component modulo 4.

The main idea for computing table $T_G[\cdot]$ by dynamic programming is to decompose G into smaller edge-disjoint subgraphs $G = G_1 \cup \cdots \cup G_k$ such that the copy sets of G can be constructed from copy sets for each of the G_i. We call such a decomposition *proper partition* if for every vertex v of G there exists an index i such that G_i contains all outgoing edges of v. Let $G_1, \ldots, G_k$ be a proper partition of G and let $\mathcal{C}_i \subseteq \mathcal{C}(G_i)$ for $i = 1, \ldots, k$. We define $\mathcal{C}_1 \otimes \cdots \otimes \mathcal{C}_k = \{C_1 \cup \cdots \cup C_k \mid C_i \in \mathcal{C}_i, i = 1, \ldots, k\}$. It is not hard to see that $\mathcal{C}(G_1 \cup \cdots \cup G_k) = \mathcal{C}(G_1) \otimes \cdots \otimes \mathcal{C}(G_k)$.

Disconnected RTGs. We start with the case that G is disconnected and consists of connected components $G_1, \ldots, G_k$, which form a proper partition of G. The main issue is to keep track of diff and cost. For an RTG G, we define $\mathcal{C}(G; d) = \{C \in \mathcal{C}(G) \mid \text{diff}(G - C) = d\}$. By Lemma 8(i) and the signature-linearity of diff, if $C_i \in \mathcal{C}(G_i; d_i)$ for $i = 1, 2$, then $C_1 \cup C_2 \in \mathcal{C}(G_1 \cup G_2; d_1 + d_2)$. This leads to the following lemma.

Lemma 9. *Let G be an RTG and let G_1, G_2 be vertex-disjoint RTGs. Then (i) $\mathcal{C}(G) = \bigcup_d \mathcal{C}(G; d)$ and (ii) $\mathcal{C}(G_1 \cup G_2; d) = \bigcup_{d'} (\mathcal{C}(G_1; d') \otimes \mathcal{C}(G_2; d - d'))$.*

By further exploiting the signature-linearity of cost, we also get $\text{cost}((G_1 \cup G_2) - (C_1 \cup C_2)) = \text{cost}(G_1 - C_1) + \text{cost}(G_2 - C_2)$, allowing us to compute the cost of copy sets formed by the union of copy sets of vertex-disjoint graphs.

Lemma 10. *Let G_1, G_2 be two vertex-disjoint RTGs and let $G = G_1 \cup G_2$. Then $T_G[d] = \min_{d'}\{T_{G_1}[d'] + T_{G_2}[d - d']\}$.*

Proof. Applying the definition of $T_G[\cdot]$ as well as Lemma 9 (ii) and Lemma 8 (i) yields

$$\begin{aligned}
T_G[d] &= \min_{C \in \mathcal{C}(G;d)} \text{cost}(G - C) = \min_{C \in \bigcup_{d'}(\mathcal{C}(G_1;d') \otimes \mathcal{C}(G_2;d-d'))} \text{cost}(G - C) \\
&= \min_{d'} \left\{ \min_{C \in \mathcal{C}(G_1;d') \otimes \mathcal{C}(G_2;d-d')} \text{cost}(G - C) \right\} \\
&= \min_{d'} \left\{ \min_{C_1 \in \mathcal{C}(G_1;d')} \text{cost}(G_1 - C_1) + \min_{C_2 \in \mathcal{C}(G_2;d-d')} \text{cost}(G_2 - C_2) \right\} \\
&= \min_{d'} \{T_{G_1}[d'] + T_{G_2}[d - d']\}. \qquad \square
\end{aligned}$$

By iteratively applying Lemma 10, we compute $T_G[\cdot]$ for a disconnected RTG G with an arbitrary number of connected components.

Lemma 11. *Let G be an RTG with n vertices and connected components $G_1, \ldots, G_k$. Given the tables $T_{G_i}[\cdot]$ for $i = 1, \ldots, k$, the table $T_G[\cdot]$ can be computed in $O(n^2)$ time.*

Tree RTGs. For a tree RTG G, we compute $T_G[\cdot]$ in a bottom-up fashion. The direction of the edges naturally defines a unique root vertex r that has no incoming edges and we consider G as a rooted tree. For a vertex v, we denote by $G(v)$ the subtree of G with root v. Let v be a vertex with children $v_1, \ldots, v_k$. How does a copy set C of $G(v)$ look like? Clearly, $G(v) - C$ contains precisely one of the outgoing edges of v, say (v, v_j). Then $Z_j = \{(v, v_i) \mid i \neq j\} \subseteq C$. Graph $G(v) - Z_j$ has connected components $G(v_i)$ for $i \neq j$, whose union we denote $G_{\neg j}$, and one additional connected component $G^+(v_j)$ that is obtained from $G(v_j)$ by adding the vertex v and the edge (v, v_j). This forms a proper partition of $G(v) - Z_j$. As above, we decompose the copy set $C - Z_j$ further into a union of a copy set $C_{\neg j}$ of $G_{\neg j}$ and a copy set C_j of $G^+(v_j)$. Graph $G_{\neg j}$ is disconnected and can be handled as above. Note that the only child of the root of $G^+(v_j)$ is v_j and hence C_j is a copy set of $G(v_j)$. For expressing the cost and difference measures for copy sets of $G^+(v_j)$ in terms of copy sets of $G(v_j)$, we use the correction terms Δ_{cost} and Δ_{diff}.

By Lemma 8 (ii), $\mathrm{diff}(G^+(v_j) - C_j) = \mathrm{diff}(G(v_j) - C_j) + \Delta_{\mathrm{diff}}(s)$, where s is the size of the *root path* $P(v_j, C_j)$ of $G(v_j) - C_j$, i.e., the size of the connected component of $G(v_j) - C_j$ containing v_j. An analogous statement holds for cost. More precisely, it suffices to know s modulo 4. Therefore, we further decompose our copy sets as follows, which allows us to formalize our discussion.

Definition 2. *For a tree RTG G with root v and children $v_1, \ldots, v_k$, we define $\mathcal{C}(G; d, s) = \{C \in \mathcal{C}(G; d) \mid |P(v, C)| \equiv s \pmod 4\}$. We further decompose these by $\mathcal{C}(G; d, s, j) = \{C \in \mathcal{C}(G; d, s) \mid (v, v_j) \notin C\}$, according to which outgoing edge of the root is not in the copy set.*

Lemma 12. *Let G be a tree RTG with root v and children $v_1, \ldots, v_k$ and for a fixed vertex v_j, $1 \le j \le k$, let $G^+(v_j)$ be the subgraph of G induced by the vertices in $G(v_j)$ together with v. Let further $G_{\neg j} = \bigcup_{i=1, i \neq j}^{k} G(v_i)$ and $Z_j = \{(v, v_i) \mid i \neq j\}$. Then*
(i) $\mathcal{C}(G; d) = \bigcup_{s=0}^{3} \mathcal{C}(G; d, s)$ *and* $\mathcal{C}(G; d, s) = \bigcup_{j=1}^{k} \mathcal{C}(G; d, s, j)$.
(ii) $\mathcal{C}(G^+(v_j); d, s) = \mathcal{C}(G(v_j); d - \Delta_{\mathrm{diff}}(s), s - 1)$.
(iii) $\mathcal{C}(G; d, s, j) = \bigcup_{d'} (\mathcal{C}(G_{\neg j}; d') \otimes \mathcal{C}(G^+(v_j); d - d', s) \otimes \{Z_j\})$.

To make use of this decomposition of copy sets, we extend our table T with an additional parameter s to keep track of the size of the root path modulo 4. We call the resulting table $\tilde{T}$. More formally, $\tilde{T}_v[d, s] = \min_{C \in \mathcal{C}(G(v); d, s)} \mathrm{cost}(G(v) - C)$. It is not hard to see that $T_G[\cdot]$ can be computed from $\tilde{T}_r[\cdot, \cdot]$ for the root r of a tree RTG G.

Lemma 13. *Let G be a tree RTG with root r. Then $T_G[d] = \min_s \tilde{T}_r[d, s]$.*

To compute $\tilde{T}_v[\cdot, \cdot]$ in a bottom-up fashion, we exploit the decompositions from Lemma 12 and the fact that we can update the cost function from $G(v_j) - C_j$ to $G^+(v_j) - C_j$ using the correction term Δ_{cost}. The proof is similar to that of Lemma 10 but more technical.

Lemma 14. *Let G be a tree RTG, let v be a vertex of G with children $v_1, \ldots, v_k$, and let $G(v_i) = (V_i, E_i)$ for $i = 1, \ldots, k$. Then with $G_{\neg j} = (V_{\neg j}, E_{\neg j}) = \bigcup_{i=1, i \neq j}^{k} G(v_i)$*

$$\tilde{T}_v[d, s] = \min_{j \in \{1, \ldots, k\}} \min_{d'} T_{G_{\neg j}}[d'] + \tilde{T}_{v_j}[d - d' - \Delta_{\mathrm{diff}}(s), (s - 1) \bmod 4] + \Delta_{\mathrm{cost}}(s).$$

For leaves v of a tree RTG G, $\tilde{T}_v[0, 1] = 0$ and all other entries are ∞. We compute $T_G[\cdot]$ by iteratively applying Lemma 14 in a bottom-up fashion, using Lemma 13 to compute $T[\cdot]$ from $\tilde{T}[\cdot, \cdot]$ in linear time when needed.

Lemma 15. *Let $G = (V, E)$ be a tree RTG with n vertices and root r. The tables $\tilde{T}_r[\cdot, \cdot]$ and $T_G[\cdot]$ can be computed in $O(n^3)$ time.*

Connected RTGs Containing a Cycle. We only give a sketch. The idea is that such an RTG contains a single directed cycle. Every copy set contains either an edge of that cycle or it contains all edges that have their source on the cycle but do not belong to the cycle. This leads to a linear number of tree instances, which we solve using Lemma 15.

Lemma 16. *Let G be a connected RTG containing a directed cycle. The table $T_G[\cdot]$ can be computed in $O(n^4)$ time.*

Putting Things Together. To compute $T_G[\cdot]$ for an arbitrary RTG G, we first compute $T_K[\cdot]$ for each connected component K of G using Lemmas 15 and 16. Then, we compute $T_G[\cdot]$ using Lemma 11 and the length of an optimal shuffle code using Lemma 7. To actually compute the shuffle code, we augment the dynamic program computing $T_G[\cdot]$ such that an optimal copy set C can be found by backtracking in the tables. An optimal shuffle code is then found by applying GREEDY to $G - C$ and adding one copy operation for each edge in C.

Theorem 3. *Given an RTG G, an optimal shuffle code can be computed in* $O(n^4)$ *time.*

Conclusion. We have presented an efficient algorithm for generating optimal shuffle code using copy instructions and permutation instructions, which allow to arbitrarily permute the contents of up to five registers. As an intermediate result, we have proven the optimality of the greedy algorithm for factoring a permutation into a minimal product of permutations, each of which permutes up to five elements. It would be interesting to allow permutations of larger size.

Acknowledgments. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).

References

1. Blazy, S., Robillard, B.: Live-range unsplitting for faster optimal coalescing. In: Languages, Compilers, and Tools for Embedded Systems (LCTES '09). pp. 70–79. ACM (2009)
2. Bouchez, F., Darte, A., Rastello, F.: On the complexity of register coalescing. In: Code Generation and Optimization (CGO '07). pp. 102–114. IEEE (2007)
3. Buchwald, S., Mohr, M., Rutter, I.: Optimal shuffle code with permutation instructions. CoRR abs/1504.07073 (2015), http://arxiv.org/abs/1504.07073
4. Caprara, A.: Sorting by reversals is difficult. In: Computational Molecular Biology (RECOMB'97). pp. 75–83. ACM (1997)
5. Farnoud, F., Milenkovic, O.: Sorting of permutations by cost-constrained transpositions. IEEE Transactions on Information Theory 58(1), 3–23 (2012)
6. Grund, D., Hack, S.: A fast cutting-plane algorithm for optimal coalescing. In: Krishnamurthi, S., Odersky, M. (eds.) Compiler Construction, Lecture Notes in Computer Science, vol. 4420, pp. 111–125. Springer Berlin Heidelberg (2007)
7. Hack, S.: Register Allocation for Programs in SSA Form. Ph.D. thesis, Universität Karlsruhe (2007), http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/6532
8. Hack, S., Goos, G.: Copy coalescing by graph recoloring. SIGPLAN Notices 43(6), 227–237 (2008)
9. Mohr, M., Grudnitsky, A., Modschiedler, T., Bauer, L., Hack, S., Henkel, J.: Hardware acceleration for programs in SSA form. In: Compilers, Architecture and Synthesis for Embedded Systems (CASES '13). ACM (2013)
10. Seress, Á.: Permutation Group Algorithms, vol. 152. Cambridge University Press (2003)