Hardware Acceleration for Programs in SSA Form

Manuel Mohr⋆, Artjom Grudnitsky⋆, Tobias Modschiedler⋆, Lars Bauer⋆, Sebastian Hack†, Jörg Henkel⋆
⋆Karlsruhe Institute of Technology {manuel.mohr,artjom.grudnitsky,lars.bauer,henkel}@kit.edu
†Saarland University hack@cs.uni-saarland.de

Abstract

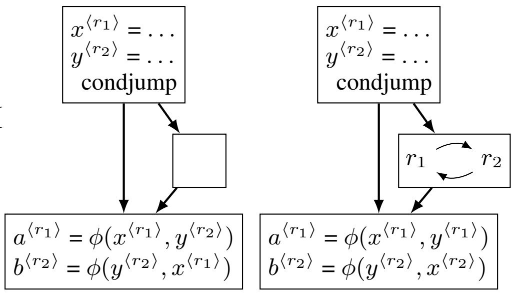
Register allocation is one of the most time-consuming parts of the compilation process. Depending on the quality of the register allocation, a large amount of shuffle code to move values between registers is generated. In this paper, we propose a processor architecture extension to provide register file permutations by which the shuffle code can be implemented more efficiently. We present compiler support to utilize this extension, an evaluation regarding performance and compilation time using the SPEC CINT2000 benchmark, as well as an analysis of area and frequency overhead of our architecture implementation. We find that using our extension, the number of executed instructions is reduced by up to 5.1% while the compilation time is unaffected.

1. Introduction

Static Single Assignment (SSA) form has become a key property of modern compiler intermediate languages [1, 8, 25]. SSA form introduces the concept of ϕ-functions that select one of their arguments depending on the control flow. While the semantics of ϕ-functions are precisely defined, ϕ-functions are a theoretical construct and must be translated into primitive machine operations during code generation. This process is often called SSA elimination or SSA destruction.

Traditionally, SSA form is destructed before register allocation to make the resulting intermediate code compatible with non-SSA-aware register allocators. However, premature SSA destruction unnecessarily constrains register allocation [19]. Research in SSA-based register allocation has lead to register allocators that directly work on intermediate code in SSA form and sustain the SSA property until after register allocation [5, 11, 21]. Here, the ϕ-functions are still present in the register allocated program.

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


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

(a) Source code. **(b)** After register allocation. **(c)** Inserted shuffle code.

Figure 1: SSA-based register allocation; $x^{\langle R \rangle}$: value x is kept in register R.

Figure 1a shows a simple program that defines two variables x and y, swaps them if some non-constant condition holds, and subsequently uses x and y. The control flow graph of the program in SSA form after SSA-based register allocation is shown in Figure 1b. We use the notation $x^{\langle R \rangle}$ to denote that the value x is kept in register R at this program point. The conditional swapping of x and y in the source program is completely encoded in the ϕ-functions. For example, the first ϕ-function selects its first argument if the left branch is taken and selects the second argument if the right branch is taken, which complies with the semantics of the original program.

However, the ϕ-functions now choose between values held in different registers. As no regular processor directly offers ϕ-instructions, the ϕ-functions must be implemented using *shuffle code* that compensates for register mismatches. In the example from Figure 1b, this means that the compiler has to insert shuffle code that swaps the contents of registers r_1 and r_2 in the second basic block. In general, shuffle code must be also inserted before instructions with register constraints if the required value is not already in the correct register.

The amount of shuffle code that has to be inserted directly depends on the quality of the copy coalescing that has been performed during register allocation. Copy coalescing tries to reduce the cost for moving values between registers as much as possible. As copy coalescing is NP-complete [6], reducing the amount of shuffle code comes at great cost in terms of compilation time. Therefore, in certain scenarios like just-in-time compilation, a high amount of shuffle code cannot be avoided.

The semantics of ϕ-functions dictate that all ϕ-functions in a basic block must be evaluated simultaneously. Hence, shuffle code consists of parallel copy operations. We can visualize a parallel copy using a *register transfer graph* (RTG). An RTG is a directed graph, where nodes represent registers and edges represent copy operations between registers. Every node has at most one incoming edge, so each register contains an unambiguous value after all copy operations have taken place. We will revisit RTGs in more detail in Section 5. In our example, the RTG that is depicted in Figure 1c states that r_1 must be transferred to r_2 and, *in parallel*, r_2 must be transferred to r_1, effectively swapping r_1 and r_2.

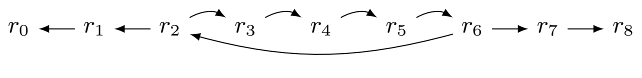


Figure 2: A more complex register transfer graph.

Generally, RTGs can be more complex, as shown in Figure 2. On regular processor architectures, RTGs must be implemented using register-register copies and, if existent, register-register swaps. For large RTGs, this can lead to a substantial amount of code being generated. Hence, it is desirable to be able to implement RTGs more concisely, ideally with a single instruction. This would require fewer instructions, and thus decrease code size and increase performance.

However, to implement an RTG in a single instruction, that instruction would need to perform several write accesses to the register file. In the example RTG from Figure 2, the value from r_6 has to be written to both r_2 and r_7. As write ports on a register file are expensive [14, 15, 32], that instruction could only be implemented as a multi-cycle instruction performing one write access per cycle, but that would give away all performance benefits.

This paper proposes an alternative that, instead of modifying the actual register contents in the register file, modifies the access to the registers similar to register renaming. For the example in Figure 1c, rather than exchanging the contents of registers r_1 and r_2, just the access to these two registers is exchanged. The read access to r_1 in the last basic block will therefore lead to a read access of r_2 in the register file. This kind of register renaming is only possible for permutations, e.g. exchanging two registers or the cycle $r_2 \rightarrow r_3 \rightarrow r_4 \rightarrow r_5 \rightarrow r_6$ from Figure 2. Permutations guarantee that the number of accessible registers is not changed and that the accessible contents are not changed. Therefore, permutations can be implemented as a kind of register renaming (more background information about permutations is summarized in Section 3). The advantage of focusing on permutations of the register names rather than exchanging the actual register contents is the reduced size of the write ports. A register address (e.g. 5 bits to distinguish 32 different registers) is typically significantly smaller than the register content (e.g. 32 or 64 bits), which directly affects the size of the write ports.

The contributions of this paper are as follows:

1. a low-latency hardware implementation that supports permutations of register names,
2. an instruction set extension to access that hardware,
3. a compiler extension to automatically and efficiently implement arbitrary RTGs, and hereby also the ϕ-function, using the extended instruction set and
4. an extensive evaluation and analysis of the complete system in real world scenarios.

After discussing related work in Section 2, we present the mathematical background for permutations in Section 3, followed by our proposed extensions to the instruction set in Section 4. Section 5 explains the compiler modifications to implement arbitrary RTGs using the extended instruction set and Section 6 discusses the necessary microarchitecture adaptions to implement the extended instruction set. The results are presented and analyzed in Section 7, and Section 8 concludes the paper.

2. Related Work

Register Allocation. The most influential approach to register allocation is graph coloring [12]. Here, variables are abstracted to nodes in the so-called interference graph. Two nodes are connected by an edge if liveness analysis judged the two corresponding variables live at the same time. A coloring of that graph then yields a correct register allocation. In terms of coalescing (i.e. the reduction of copy instructions in the code), two copy-related nodes in the interference graph are fused. This can increase the register demand of the program which in turn can lead to additional spill code. Hence, the coalescing approach of original graph coloring is called *aggressive* coalescing. Chaitin also showed that for every undirected graph there exists a program, which has that graph as its interference graph [12]. Hence, graph coloring register allocation is NP-hard.

Over the last decades, this technique has been improved and different coalescing techniques have been proposed. Briggs et al. [10] derived criteria for *conservative* coalescing, which means that coalescing will never trade a copy for a spill. Park and Moon [26] proposed optimistic coalescing, which is a conservative technique that tries to undo aggressive coalescing in the case a spill was introduced because of a coalesced copy.

Recently, different articles proposed performing register allocation on SSA-form programs [5, 11, 21]. There, the interference graph falls into the class of chordal graphs, which makes it colorable in polynomial time. The difference to traditional graph coloring allocation is that the ϕ-functions are still present *after* register allocation [21]. The ϕ-functions in register-allocated SSA programs correspond to parallel copy operations and can be implemented by a sequence of swap and copy instructions. Rideau et al. [28] give a formal proof for the implementation correctness of parallel copies.

Instead of implementing the parallel copies at the place where the ϕ-function is (usually the end of the preceding basic blocks), Bouchez et al. [4] proposed a technique to move the parallel copy to other places such that its implementation involves fewer copies. In our compiler, we use a faster but less sophisticated technique, which leaves more parallel copies in the code. Essentially, we try to hoist parallel copies inside a block to a location with less register pressure. However, this technique is not a contribution of this paper and it was enabled during all measurements presented in Section 7.

In SSA-based register allocation, it is up to the register assignment or a coalescing post pass to find an assignment that involves as few copies as possible. This problem is again NP-hard even on SSA-form programs [6, 19]. Various coalescing techniques for SSA-based register allocation have been proposed. Pereira et al. [27] and Bouchez et al. [7] propose novel conservative criteria for node coalescing. Hack and Goos [20] introduce recoloring to improve a found coloring by trying to assign two copy-related nodes the same color and [17] present an efficient ILP-based algorithm. Braun et al. [9] and Colombet et al. [13] present biasing techniques for the register assignment phase: Usually, the allocator chooses one register out of a list of free registers. By biasing this choice, those allocators try to pick the same registers for copy-related variables in the first place instead of relying on a post pass like recoloring. Since biasing usually produces colorings of inferior quality compared to more heavyweight techniques like recoloring or even optimal ILP-based ones, in this paper we propose to add hardware support to alleviate some of the overhead of parallel copies.

Register Renaming. The register file permutation concept presented here is a type renaming for register addresses. *Register renaming* is also used to enable out-of-order processing in modern CPUs. [29] provides an overview on register renaming background and techniques. The main conceptual difference between permutation and register renaming is that register renaming is transparent to software, being a purely hardware-controlled technique, while permutations are explicitly controlled by software.

Permutation in other ISAs. Permutation of values is supported in SIMD extensions of some existing instruction set architectures, e.g. Intel x86 [23] and PowerPC [16]. x86 offers `PSHUFB` (Packed Shuffle Bytes) as part of SSE3, which permutes bytes in a 256-bit register, with the permutation defined in a second operand register. Advanced Vector Extension (AVX) introduced the `VPERM*` instructions which do not perform in-place permutations, but write the permuted value into a destination register. In addition, `VPERM*` instructions allow value duplication. The PowerPC AltiVec extension offers the `vperm` instruction, which extracts bytes from 2 128-bit source registers and arranges them according to a user-definable mask into a 128-bit destination register. As for `VPERM*`, value duplication is permitted.

Both ISAs allow permutation only on values *within* one (or two) registers, but not between registers. Furthermore, the instructions are limited to special registers for SIMD processing. For implementing ϕ-functions, permutation of multiple general-purpose registers is required, thus the above mentioned instructions cannot be used.

3. Mathematical Background of Permutations

Permutations can be expressed in cycle notation. In Equation (1), the two-line notation of permutation σ on the left – with the first row listing the elements x_i and the second row the images $\sigma(x_i)$ onto which they are mapped – is equivalent to the cycle notation on the right.

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 5 & 2 & 6 & 1 & 4 \end{pmatrix} = (1\,3\,2\,5)(4\,6) \tag{1}$$

Cycle notation describes a permutation σ as a product of cycles (Equation (2)). A cycle c_i is constructed by starting with an element x_k, which is not part of any other cycle, and appending the image of the element $\sigma(x_k)$. The σ operator is applied on the last member of this cycle, until $x_k = \sigma^d(x_k)$, which is not appended to the cycle. The cycle length $|c_i|$ is d.

$$\begin{array}{l} \sigma = \prod_i c_i \\ c_i = \big(x_k\ \sigma(x_k)\ \sigma^2(x_k) \dots \sigma^{d-1}(x_k)\big), \forall j \neq i : x_k \notin c_j \end{array} \tag{2}$$

Elements that are not part of any cycle are fixed points of the respective permutation, i.e., element x_f and its image $\sigma(x_f)$ are equal. A cycle c of length $d > 2$ can be further decomposed into two cycles c_1, c_2 with a combined length $|c_1| + |c_2| = d + 1$ as shown in Equation (3).

$$\begin{array}{rcl} c_i & = & \big(x_k\ \sigma(x_k)\ \sigma(\sigma(x_k))\ \cdots\ \sigma^{d-1}(x_k)\big) \\ & = & \big(x_k\ \cdots\ \sigma^{d_1}(x_k)\big)\big(\sigma^{d_1}(x_k)\ \cdots\ \sigma^{d-1}(x_k)\big) \end{array} \tag{3}$$

For example, cycle $(1\,3\,5)$ has a length of 3 and can be decomposed into cycles $(1\,3)$ and $(3\,5)$ with a combined length of 4.

As cycles allow for a compact representation of permutations and any permutation can be decomposed into cycles, our permutation instruction takes cycles as arguments.

4. Extensions to the Instruction Set Architecture

Permuting the register file is an extension to the base processor capabilities. We have extended the SPARC V8 ISA [30] with new *permutation instructions* that apply permutations to the register file. Permutations can be expressed in different notations (see Section 3). We use the cycle notation due to its compactness.

For register file permutation, the maximum length of a cycle is the amount of logical registers (32 for our implementation). However, instruction width limits the size of a cycle that can be expressed by a single instruction. The opcode uses o bits of the instruction word, leaving $n - o$ bits for encoding a permutation, with n being the instruction width. For 32 visible registers, $\lceil \log_2(32) \rceil = 5$ bits are required to identify one register (i.e. encode one element of the cycle). In our implementation for SPARC V8 we need 7 bits for the opcode, leaving us with 25 bits for encoding the permutation. This allows us to encode cycles with a length of up to 5 elements as the immediate of the permutation instruction.

As stated in Section 1, the amount of shuffle code, and therefore also the average size of an RTG depends on the quality of copy coalescing. Using programs from the CINT2000 benchmark suite, we found that for realistic coalescing schemes, the average size of an RTG was 4.6 or less. Small RTGs with less than 5 nodes were far more frequent than RTGs with more than 5 nodes. Therefore, the ability to encode permutations of maximum size 5 matches the needs when implementing RTGs. We will present more elaborate results on the properties of RTGs in Section 7.2.

An alternative approach for encoding permutations is storing the permutation in a register. However, this would only increase the maximum cycle length to $32/\lceil \log_2(32) \rceil = 6$-element cycles, while sacrificing one register.

We have extended the SPARC V8 ISA with two instructions for permuting register files: (i) `permi5` applies a 5-cycle permutation and (ii) `permi23` applies a 2-cycle and a 3-cycle permutation, with both cycles completely independent (i.e. no element from one cycle is part of the other cycle). Both instructions use the same format, given in Figure 3. The instructions have 5 operands, a, b, c, d and e. Due to limitations of the free opcode space, a cannot be encoded as 5 consecutive bits, but has to be split into the upper 3 bits a_1 and the lower 2 bits a_2. For `permi5`, each argument corresponds to a member of the 5-cycle; For `permi23`, a and b encode the 2-cycle, while the 3-cycle is encoded with c, d, e. Cycles shorter than 5 are

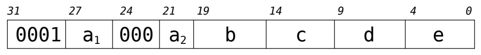


Figure 3: Format for permutation instructions, implemented for the SPARC V8 ISA.

encoded by repeating the last member of the cycle until `permi5` has 5 elements, e.g.:

```
permi5 %r2, %r3, %r3, %r3, %r3
```

would encode a swap of %r2 and %r3. `permi23` can be used to encode two 2-cycles using the same technique. The `permi5` and `permi23` instructions are discerned by the hardware by comparing the first two operands. If the register numbers are in ascending order (i.e. $a < b$), the instruction is interpreted as `permi5`, otherwise as `permi23`. In the remainder of the paper we will use `permi` when referring to either permutation instruction.

5. Code Generation for Register Transfer Graphs

A *register transfer graph* (RTG) is a directed graph $G = (V, E)$, where a node $v \in V$ represents a register and an edge $(v, v') \in E \subseteq V \times V$ represents a copy operation that copies the value from source register v to destination register v'. All copy operations in an RTG are assumed to be performed *in parallel*. Therefore, each node in the graph has at most one incoming edge, because the register content is undefined if multiple concurrent copy operations write to the same destination register. However, a node can have multiple outgoing edges, which means that the register value is duplicated.

$$\begin{tikzcd} r_0 \arrow[r, bend left] & r_1 \arrow[r, bend left] & r_2 \arrow[ll, bend left] & r_3 \arrow[r] & r_4 \arrow[r] & r_5 & r_6 \arrow[r, bend left] & r_7 \arrow[l, bend left] \arrow[r] & r_8 \end{tikzcd}$$

Figure 4: Example register transfer graph.

Figure 4 shows a possible register transfer graph. In this case, the graph has three connected components, i.e. three register transfer subgraphs with disjoint node sets and disjoint edge sets. Each component of an RTG is itself a valid RTG. In the example, the left component, involving the registers r_0, r_1 and r_2 expresses a cyclic copying of values. In the following, we will call such components *cycles*. A register transfer graph $G_1 = (V_1, E_1)$ is a cycle iff it is connected and each node $v \in V_1$ has exactly one incoming edge and exactly one outgoing edge. The middle component shows a *chain*. A register transfer graph $G_2 = (V_2, E_2)$ is a chain iff there is exactly one node $v_S \in V_2$, the start node, with no incoming edges, exactly one node $v_E \in V_2$, the end node, with no outgoing edges and there exists a path of length $|V_2|$ from v_S to v_E. The component on the right consists of a subgraph that swaps the contents of registers r_6 and r_7, and a subgraph that copies r_7 to r_8. However, this graph does not consist of a cycle and a chain because the two subgraphs are intertwined and do not comply with the definitions of cycle and chain. It is, however, still a valid RTG because there does not exist a node with more than one incoming edge.

5.1 Implementing RTGs on Regular Machines

When implementing an RTG, special care has to be taken that the parallel copy semantics are preserved. On traditional machines, implementing a given RTG G works as follows [19]:

1. While there is a node n with no outgoing edges: There must be exactly one edge (n', n) with $n \neq n'$ and we can insert a register-register copy $n' \to n$, because the value in register n is not needed anymore. Remove the edge (n', n) from G's edge set and repeat step 1.
2. Now G is either empty or consists solely of cycles. Cycles of length 1 (self-loops) do not generate any instructions. All cycles of length 2 or greater can be implemented as follows:
   - If there is a free register r_t, a cycle $(r_1, \dots, r_k)$ can be implemented by $k + 1$ copies following the scheme $r_k \to r_t, r_{k-1} \to r_k, \dots, r_t \to r_1$.

- If there is no free register, a cycle of length k can be decomposed into $k-1$ transpositions, which can be implemented using $k-1$ register-register swap instructions. If the instruction set does not offer a swap instruction, we can implement each transposition using 3 `xor` instructions, requiring $3\cdot(k-1)$ `xor` instructions for the complete cycle.

5.2 Implementing RTGs Using Permutation Instructions

If permutation instructions are available, they can be exploited during code generation for RTGs. To recap the capabilities of our permutation instructions: the `permi5` instruction can implement a single cycle of maximum size 5, i.e. a cycle of size 2, 3, 4 or 5, while the `permi23` instruction can implement two cycles, the first of which must be a cycle of size 2 and the second cycle can either be of size 2 or 3.

The central questions at this point are:

1. Which RTGs can be implemented using only permutation instructions and no additional copies?
2. How can the remaining RTGs be converted into a form that can then be implemented using permutation instructions and as few additional copies as possible?

The first question can be answered concisely: every RTG that only consists of cycles can be directly implemented using permutation instructions. We say that such RTGs are in *permutation form*, as a product of cycles defines a permutation. Thus, finding a valid implementation for a graph in permutation form using only the `permi` instructions is straightforward: Cycles of maximum size 5 can be directly translated into one `permi5` instruction, while larger cycles can be decomposed into a product of cycles of maximum size 5 and then expressed using a series of `permi5` instructions. In general, this does not generate a short instruction sequence, because it does not make use of the `permi23` instruction. We will revisit the problem of finding a good solution for expressing a given RTG in permutation form using permutation instructions in Section 5.2.2.

5.2.1 Conversion into Permutation Form

This leaves the second question: how can the remaining RTGs be converted into permutation form? First, let us look at chains. Chains can be easily converted into cycles. Take the chain $r_3 \to r_4 \to r_5$, the middle component from Figure 4, as an example. Note that because there is no self-loop $r_3 \to r_3$, the value from r_3 is only needed in r_4 after the parallel copy operation. Therefore, after the parallel copy has finished the content of register r_3 is irrelevant to further program execution and r_3 can be overwritten with an arbitrary value. This allows us, as shown in Figure 5, to add the artificial edge $r_5 \to r_3$ to the graph, thereby converting it to a cycle and reducing it to the aforementioned case.

$$\begin{tikzcd}
r_3 \arrow[r] & r_4 \arrow[r] & r_5 & \to & r_3 \arrow[r, bend left] & r_4 \arrow[r, bend left] & r_5 \arrow[ll, bend left, dashed]
\end{tikzcd}$$

Figure 5: Converting a chain into a cycle by adding the artificial edge $r_5 \to r_3$.

However, the rightmost component of the register transfer graph from Figure 4 is neither a cycle nor a chain. The fundamental problem with this class of graphs is that they duplicate values. In the example, the value from r_7 is written to both r_6 and r_8. Permutations are injective functions, so a duplication of values is inherently impossible. Therefore, the RTG must be modified in a way that allows the conversion of at least a part of the graph into permutation form. The general goal here is to make the resulting graph as big as possible and to preserve existing cycles in the RTG to maximize the potential benefit of using permutation instructions.

We will convert the original RTG into an RTG in permutation form and a list of copy instructions that must be executed after the instructions that implement the graph itself. At each node in the graph that has more than one outgoing edge, the conversion must keep exactly one of the edges and express all other edges by copy instructions.

$$\begin{tikzcd}
r_6 \arrow[r, bend left] & r_7 \arrow[l, bend left] \arrow[r] & r_8 & \to & r_6 \arrow[r, bend left] & r_7 \arrow[l, bend left] & + & r_6 \arrow[r] & r_8
\end{tikzcd}$$

Figure 6: Example for conversion of a graph, which is neither a cycle nor a chain, into a graph in permutation form and a list of additional copies.

Consider the example in Figure 6, which is the right component of the RTG from Figure 4. As outlined before, we can either keep edge (r_7, r_6) or edge (r_7, r_8), and express the other one with a copy, to obtain an RTG in permutation form. To prevent splitting the cycle, we express (r_7, r_8) with a copy instruction in this case. However, by doing this, we would not preserve the semantics of the parallel copy because the copy instruction for (r_7, r_8) would be executed after the instruction that implements the cycle and thus after r_7 is modified. Therefore, we have to copy r_6 to r_8 *after* the permutation of values has taken place. Restoration of values is always possible because permutations are injective functions, hence all values are still accessible after a value permutation, even though they are held in different registers.

In general, each connected component can contain at most one cycle, as otherwise there would be a node with more than one incoming edge. Each node of the cycle can in turn be the root of a *tree*. Because we preserve the cycle, as described in the previous step, we must now process the tree. Our goal here is, as mentioned before, to require as few copies as possible and thus make the resulting RTG in permutation form as big as possible.

$$\begin{tikzcd}
 & r_1 & & r_5 \arrow[r] & r_8 & & r_0 \arrow[r, bend left] & r_2 \arrow[r, bend left] & r_3 \arrow[r, bend left] & r_5 \arrow[r, bend left] & r_8 \arrow[llll, bend left, dotted] \\
r_0 \arrow[ur] \arrow[r] & r_2 \arrow[r] \arrow[dr] & r_3 \arrow[ur] \arrow[r] & r_6 & & \to & r_4 \arrow[r, bend left] & r_7 \arrow[l, bend left, dotted] & & & \\
 & & r_4 \arrow[r] & r_7 & & & & & + & & \\
 & & & & & & r_2 \arrow[r] & r_1 & & r_5 \arrow[r] & r_6 \\
 & & & & & & & & r_3 \arrow[r] & r_4 &
\end{tikzcd}$$

Figure 7: Conversion of a tree-shaped RTG into a graph in permutation form and a list of additional copies.

We use the following heuristics to transform a tree-structured RTG G into another RTG G' in permutation form and an associated list of copy instructions: at each node r in G that has more than one outgoing edge, we choose to keep the edge (r, r') that is part of the longest path starting from r. Consider the example from Figure 7. There are three nodes that have more than one outgoing edge: r_0, r_2 and r_3. For example, at node r_2 we can keep either (r_2, r_3) or (r_2, r_4). In this case, we keep (r_2, r_3) because the path (r_2, r_3, r_5, r_8) is longer than the path (r_2, r_4, r_7).

5.2.2 Decomposition into Cycles

After this conversion is done, the resulting RTG G' will only consist of cyclic components. In other words, the graph now represents a register permutation, expressed as a product of cycles. This permutation must now be implemented with as few `permi` instructions as possible. To start, let us look at an example where generating optimal code is not immediately obvious.

$$\begin{tikzcd}
r_0 \arrow[r, bend left] & r_1 \arrow[r, bend left] & r_2 \arrow[ll, bend left] & r_3 \arrow[r, bend left] & r_4 \arrow[r, bend left] & r_5 \arrow[r, bend left] & r_6 \arrow[r, bend left] & r_7 \arrow[r, bend left] & r_8 \arrow[lllll, bend left]
\end{tikzcd}$$

Figure 8: A sample RTG after conversion into permutation form.

Figure 8 shows an RTG consisting of a 3-cycle and a 6-cycle. A simple code generation approach considers the graph components in separation. Thus, it would first generate code to implement the 3-cycle, which can be done with a single `permi5` instruction. The 6-cycle can be split up into a 5-cycle and a 2-cycle, thereby requiring two `permi5` instructions. However, this is not optimal, as the 3-cycle and the 2-cycle that has been split off the 6-cycle can be implemented by a single `permi23` instruction. Therefore, it is crucial not to examine components in separation but to consider all of them simultaneously.

We propose a greedy algorithm for finding a short `permi` instruction sequence that implements a given RTG in permutation form. The idea behind the algorithm is as follows. Implementing an n-cycle requires $n-1$ transpositions. The `permi5` instruction implements a cycle of maximum length 5, so it implements at most 4 transpositions. The `permi23` instruction implements a cycle of length 2 and a cycle of maximum length 3, a total of at most $1+2=3$ transpositions. In this sense, the `permi5` instruction is more powerful than the `permi23` instruction, because it can implement one additional transposition.

This means that if there is a cycle of length 5 or more, it is always optimal to emit as many fully-utilized `permi5` instructions as possible until the cycle size has been reduced to less than 5. It is also optimal to implement 4-cycles with one `permi5` instruction, even if this leaves one transposition unused. Initially, it seems worthwhile to split a 4-cycle c_4 into a 2-cycle c_2 and a 3-cycle c_3 and to combine them with other small cycles using the `permi23` instruction. For example, we could implement c_2 and another 3-cycle c_3' using a `permi23` instruction, leaving us with c_3. However, we now have implemented one 4-cycle using one instruction, because we started with c_4 and c_3' and ended up with c_3, so we could have also used a `permi5` for the 4-cycle. The same reasoning can be applied for c_3. Hence, only if multiple 2-cycles or 3-cycles are available at the same time, the `permi23` instruction has to be used.

Algorithm 1 Greedy algorithm for finding a short `permi` instruction sequence to implement an RTG in permutation form.

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

    # First phase: only emit permi5 instructions
    while longs ≠ []:
        cycle ← longs.take()
        while cycle.length() ≥ 4:
            (cycle', remainder) ← split(cycle)
            insns.add(Permi5(cycle'))
            cycle ← remainder
        if cycle.length() > 0:
            shorts.add(cycle) # Remember remainder

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

Algorithm 1 shows the pseudo code of our greedy algorithm to find a short sequence of `permi` instructions that implement a given RTG in permutation form. We assume that 1-cycles (self-loops) have been filtered out. The list operation `take()` is a destructive operation that removes a list element and returns it. The algorithm consists of an initialization step and two phases. During initialization, the algorithm sorts all cycles into two lists: *long* cycles of length 4 or more and *short* cycles of lengths 2 or 3. During the first phase, the algorithm generates `permi5` instructions for each long cycle until its length has been reduced to 3 or less. The remaining cycle (if any) is added to the list of short cycles. After the first phase has finished, the list of long cycles is empty.

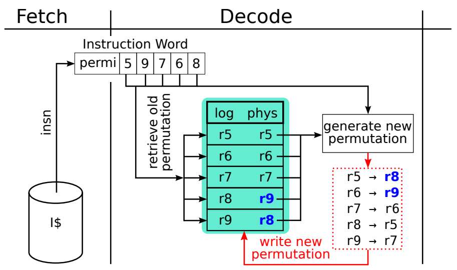


Figure 9: Defining a new permutation using the `permi5` instruction.

The second phase aims to fully utilize the `permi23` instruction by trying to implement multiple short cycles using one instruction. If there is at least one 3-cycle left, we already found a perfect match for the second part of our `permi23` instruction. Therefore we want to utilize the ability to swap two additional registers as well, if possible. If there is a 2-cycle, it is certainly optimal to use it. If there are no 2-cycles but other 3-cycles left, we split a 3-cycle into two 2-cycles, use one to fill our `permi23` instruction and remember the other one. Finally, if the 3-cycle was the last cycle left to implement, we generate a `permi5` instruction and are done. If we do not have 3-cycles but only 2-cycles left, we use a `permi23` if we have at least two 2-cycles and otherwise a `permi5` for the last 2-cycle.

5.2.3 Time Complexity

We will analyze the worst case complexity of the algorithm for an input RTG with n nodes. As each node has at most one incoming edge, n is also an upper bound for the number of edges. The conversion step needs to find all cycles in the graph, which can be done in $\mathcal{O}(n)$ using, e.g., Tarjan's SCC algorithm [31]. Additionally, the longest path starting from each node in the resulting trees must be determined, which can also be done in $\mathcal{O}(n)$ using a depth-first search.

The greedy decomposition algorithm generates one instruction per iteration, thereby reducing the input size by at least 1 each iteration. Hence, n is an upper bound for the number of iterations. As splitting a cycle only takes constant time, the amount of work done per iteration is in $\mathcal{O}(1)$, and thus the whole decomposition is in $\mathcal{O}(n)$. In total, this leads to a linear worst case complexity.

6. Architecture Support for Register Permutation

6.1 Fundamental Pipeline Modifications

We present our implementation for hardware-based SSA support using an in-order SPARC V8 architecture [30], however, the concept is not restricted to this particular ISA or microarchitecture. The underlying processor for the presented implementation is a Gaisler LEON 3 [2], which uses an in-order 7-stage pipeline. An instruction is retrieved in the *Fetch* stage, decoded in the *Decode* stage, and operands are retrieved from the register file in the *Register* stage. For arithmetic and branch instructions, the respective operation is done in the *Execute* stage, while memory operations (e.g., Load & Store) perform their operation in the *Memory* stage. The *Exception* stage is used for interrupt and trap handling, and the result of an instruction (if any) is written back to the register file in the *Writeback* stage. In SPARC V8 architectures the register file is organized in multiple *register windows*. While the register file may have enough space for 136 entries in an implementation with 8 windows, only 32 registers are visible at a time, defined by the *current window pointer*. The current window pointer is incremented or decremented by SPARC

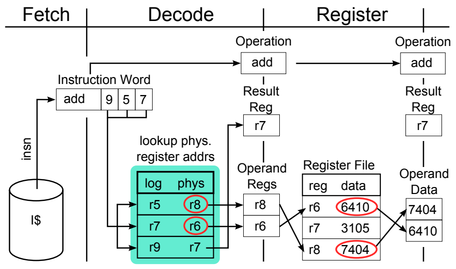


Figure 10: Executing an add instruction on a permuted register file.

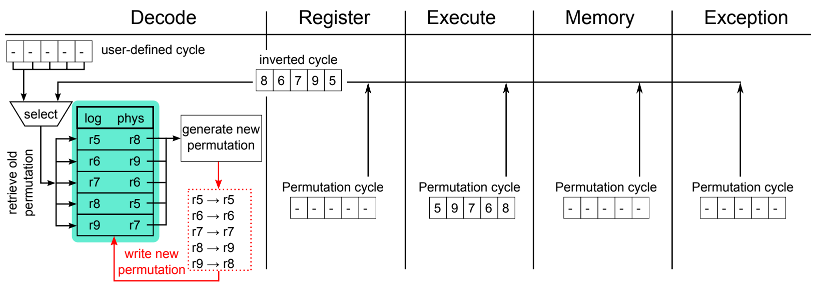


Figure 11: Implementation of permutation reversion during trap detection.

V8 instructions. Details about register windows can be found in the SPARC V8 standard [30].

To support register file permutation the main extension is the *permutation table*, which stores the current logical-to-physical mapping of register addresses for all registers (i.e. 136 entries for 8 register windows used in our implementation). This corresponds to a permutation written in two-line notation (left part of Equation (1) in Section 3).

Figure 9 shows how a new permutation is applied to the register file using the `permi5` instruction (similar for `permi23`). The permutation table is initialized with the identity (only registers $r_5 - r_9$ are shown), except for r_8 and r_9 which are exchanged. The process consists of four steps, all of which happen in the *Decode* stage:

1. The instruction decoder recognizes a `permi5` or `permi23` instruction and extracts the 5 operands that define the cycles from the instruction word.
2. As permutations are applied onto already existing permutations, first the existing permutation is read from the permutation table.
3. Permutation instructions define a permutation on the current window, thus the current window pointer is used to filter the entries of the current window from the existing permutation that was read in the previous step.
4. The cycle defined by the permutation instruction is applied to the existing permutation and the resulting permutation is written back to the permutation table.

The example in Figure 9 shows the resulting new permutation that is written to the permutation table. According to the `permi5` instruction, r_5 should become r_9, but as in the existing permutation in the permutation table r_9 is currently in r_8, r_5 becomes r_8. Figure 10 presents the pipeline activities of an `add` instruction immediately following this `permi5` instruction. It shows how the existing permutation is applied to all register addresses before accessing the register file.

```
permi5 %r5, %r9, %r7, %r6, %r8
add %r9, %r5, %r7 ; Note: %r7 is destination register
```

Permutation instructions do not induce read-after-write hazards in the pipeline, thus no extension of the pipeline forwarding logic is required for permutation support. This is due to permutation instructions *committing* their changes in the *Decode* stage, thus once the following instruction (`add` in the example) is in the *Decode* stage one cycle later, the permutation table has already been updated with the new permutation. We call this characteristic *early committing*.

At system reset the permutation table is initialized with the *identity* permutation, i.e. the physical address of a register is the same as its logical address. Subsequent permutation instructions change the permutation table. Permutations are transparent to the operating system (OS), so OS code for context switches does not need to be modified. For instance, if the OS wants to save r_5 from a task (to restore it later), it will actually access the physical register where the value of r_5 is currently stored (r_8 in Figure 10). When restoring r_5 later, it may be written to a different register, however, an access to r_5 will provide the same data that was initially saved.

6.2 Exception Handling — Background and Issues

The architecture outlined in the previous section is capable of executing programs using permutation instructions, unless traps are encountered during execution. The SPARC V8 standard specifies 3 categories of traps: *precise traps* are induced by particular instructions, e.g. unknown instructions, trap-on-condition instructions or instructions causing a register window overflow or underflow. *Deferred traps* are caused by floating point and co-processor instructions and become visible after the instruction that caused them has committed. *Interrupting traps* are caused by external interrupts, e.g. timer interrupts or IO components notifying the processor that a buffer is full. A bare metal program that does not use any IO components and uses the register window in a way that does not incur over- or underflows (e.g. by compiling the program in a way that does not change the register window during function calls), can be executed without encountering any traps. However, when executing on a multi-tasking OS, the program is likely to be interrupted, e.g. by the timer used to periodically invoke the OS scheduler, by the page fault handler, by interrupts caused by peripherals, etc.

In the underlying architecture, traps are handled in the *Exception* stage – the following issues would not occur, if trap handling was performed in the earlier *Decode* stage.

After the trap handler code has finished, the program counter and next program counter are restored and regular program execution is resumed. It has to be ensured that instructions are not executed twice, i.e. instructions already in the pipeline before the trap is detected must not first proceed through the pipeline at the start of the trap handler and then be executed again after the old program counter is restored and the corresponding instruction is reloaded. Therefore, all instructions already in the pipeline before the *Exception* stage at the time of trap detection are automatically annulled by the pipeline. This ensures that they are executed only once, after trap handling has finished, and they are loaded into the pipeline again once the old program counter is restored.

Due to their early committing characteristic, the permutation instructions cannot be handled using the mechanism described above, as they already modify the permutation table in the decode stage. We differentiate between 3 cases for handling permutation annulling during a trap:

- Permutations in the *Fetch* stage are flagged with an *annul* bit, which inhibits updating the permutation table in the *Decode* stage.
- Permutations in the *Decode* stage are notified by a *cancel* signal, which is sent at the time a trap is detected in the *Exception* stage. In this case, the new permutation is created, but not written back to the permutation table.
- Permutations in the *Register*, *Execute*, *Memory* and *Exception* stages require special handling, as they can no longer be annulled or cancelled, instead their effects need to be *reverted*. We discuss the solution for this case in Section 6.3.

Register	Execute	Memory	Exception
`permi` σ_4	`permi` σ_3	`permi` σ_2	`permi` σ_1

Figure 12: Traps require reverting of up to 4 permutations, depending on pipeline state.

It is imperative to handle all 3 cases, otherwise a permutation will be applied to the register file twice, and registers will not contain the values the program would expect, leading to a program crash.

6.3 Concept for Exception-safe Permutation

Assuming the permutation state ρ of the register file, a permutation σ that is applied on top of ρ can be reverted by applying the inverse permutation σ^{-1}:

$$\sigma^{-1}(\sigma(\rho)) = \rho \tag{4}$$

This allows reversion of all permutations that need to be annulled due to a trap, by applying inverse permutations in the reverse order the original permutations were applied. Assuming the pipeline state at the time of a trap shown in Figure 12, four inverse permutations need to be applied in the reverse order to restore the original state ρ:

$$\sigma_1^{-1}\sigma_2^{-1}\sigma_3^{-1}\sigma_4^{-1}(\sigma_4\sigma_3\sigma_2\sigma_1(\rho)) = \rho \tag{5}$$

The implementation of this concept requires that the arguments (i.e. the cycle) of a permutation instruction are propagated through the pipeline up to the *Exception* stage. This requires 4 additional 25-bit wide pipeline registers and 4 corresponding 1-bit registers, which indicate whether the instruction was a permutation. The *Exception* stage is extended to check whether at least one of the instructions in the *Register*, *Execute*, *Memory* or *Exception* stages was a permutation. If no permutations were detected in these stages, exception handling continues as usual, otherwise the pipeline is halted and each of the mentioned stages is checked for a permutation in the order *Register* $\rightarrow$ *Execute* $\rightarrow$ *Memory* $\rightarrow$ *Exception*. If a permutation is detected, an inverse permutation is computed and applied to the permutation table like a regular permutation. One inverse permutation can be computed and applied per cycle, thus permutation inversion can take up to 4 cycles per trap. Implementing inversion as a multi-cycle operation is necessary to prevent increasing critical path length (and thus reducing processor frequency). Inverse permutations can be generated by reversing the cycle of the original permutation (i.e. reversing the order of the arguments of the permutation) and applying the resulting permutation. The extensions for permutation reversion during trap occurrence are shown in Figure 11 for an example where one permutation is reverted. The hardware extensions for exception-safe permutations described in this section are implemented in our FPGA prototype as well.

7. Experimental Evaluation

Our experimental evaluation consists of three parts: First, we analyze the structure of the RTGs in our test inputs and investigate the runtime of our code generation approach. Second, we determine the quality of the executables produced by our code generation approach in terms of precise dynamic instruction counts and then validate these numbers by measuring the actual runtime of the same executables on our hardware prototype. Third, we discuss the impact of permutation reversion and present an area and frequency overhead analysis for our hardware prototype implementation.

7.1 Setup

We have implemented the code generation strategy described in Section 5 in libFIRM [8]. This compiler features a mature SPARC backend and multiple completely SSA-based register allocators and copy coalescing schemes. As compiler input, we used the test programs contained in the integer part CINT2000 of the CPU2000 benchmark suite [22]. We excluded the program `252.eon` from the measurements because the frontend does not support C++ code.

All compile time measurements were performed on an Intel Core i7 workstation with 3.4 GHz and 16 GB RAM using Linux kernel 3.5. To measure the quality of the generated code, we modified QEMU [3] to count the number of executed instructions and to support our ISA extension consisting of the `permi` instructions. Using QEMU, we were able to obtain precise dynamic instruction counts for the generated executables. All programs were compiled in soft-float mode because our prototype did not have an FPU.

To validate the results acquired from QEMU, we conducted runtime measurements on an FPGA prototype implementation of a CPU supporting our proposed instruction set extension as described in Section 6, in the following called LEON3-P. The Gaisler LEON 3 CPU served as a basis for this prototype. We synthesized a LEON 3 System-on-Chip design for the Xilinx Virtex-5 based ML509 evaluation board, with the CPU configured with 32 kB instruction cache, 32 kB data cache, 8 register windows, no FPU and a hardware multiplier. We used a Buildroot Linux (kernel version 2.6.36) distribution for benchmarking.

To test our architecture extension with a varying number of RTGs and RTGs of varying complexity, we used four different copy coalescing strategies, ordered from best to worst coalescing quality.

ILP An integer linear programming-based copy coalescer [17]. This produces RTGs with minimal cost according to the cost model. The cost incurred for a parallel copy is the number of unequally assigned registers multiplied by the estimated execution frequency of the parallel copy. Note that the number of unequally assigned registers is an estimate for the number of copy and swap operations that have to be generated for the parallel copy. In our experiments, we set the ILP solver's timeout to 5 minutes per instance of the coalescing problem. If the time limit was exceeded, we used the best solution found so far. Note that exceeding the time limit does not imply the non-optimality of the found solution. In some cases, although the solution is optimal, the solver cannot prove this fact in time. The solver used in these experiments was Gurobi 5.10 [18].

Recoloring A recoloring approach, which is currently one of the best conservative coalescing heuristics [20], resulting in RTGs with slightly higher costs.

Biased A biased coloring approach that yields good coalescing results while offering very fast allocation [9]. For our benchmarks, we disabled the initial preference analysis. In this configuration, the approach is highly suitable for just-in-time compilation scenarios. The generated code contains RTGs of higher cost than with the recoloring approach.

Naive This approach does not perform any sophisticated copy coalescing at all. Except for trying to avoid copy instructions because of register constraints, no effort is made to coalesce copies. In general, this results in RTGs with high costs.

For each coalescing strategy, we inspected the properties of typical register transfer graphs occurring in our test programs to estimate the potential benefit of using permutation instructions. Furthermore, for each of the four coalescing strategies, we tested two compiler configurations: one that generated permutation instructions using the code generation strategy presented in Section 5 and one that emitted regular SPARC code. For each of the resulting eight compiler configurations, we measured the compilation time and the quality of the generated code.

7.2 Register Transfer Graph Properties

The number and properties of RTGs are directly dependent on the used coalescing strategy, which tries to minimize the cost of RTGs according to a cost model. For ease of presentation, we will use the number of RTGs and their average size as an approximation for the costs assigned to the RTGs. In general, the number and sizes of RTGs and their costs are highly correlated. For each coalescing strategy, we analyzed the number and average size of RTGs over all programs of the `CINT2000` benchmark suite. Moreover, we checked what percentage of RTGs do not duplicate any values, i.e. can be implemented only with our `permi` instructions and without additional copy instructions.

Coalescer	Number of RTGs	Average size	No value duplication
ILP (best)	77 783	2.9	74%
Recoloring	78 194	2.9	74%
Biased	178 812	4.6	54%
Naive (worst)	185 035	6.6	89%

Table 1: Register transfer graph properties.

Table 1 show that the number of RTGs as well as the average complexity of an RTG, represented by its number of nodes, increase with decreasing coalescing quality. Furthermore, depending on the coalescing scheme, between about half and almost 90% of the RTGs did not duplicate any values, i.e. did not need additional copy instructions. For the RTGs that did need additional copies, on average 1.26 copies per RTG were needed for the ILP, the recoloring and the naive coalescing, and 1.99 copies per RTG were needed for the biased coalescing approach. This means that the vast majority of RTGs already are in permutation form or very close to it. Thus, few additional copy instructions must be inserted during the conversion step presented in Section 5.2.1, and most of the work can be done using only permutation instructions.

7.3 Compile Time

We measured the runtime of our code generation approach described in Section 5 compiling the entire `CINT2000` benchmark set and compared it to the default version implemented in libFIRM.

	Default [ms]	Our code gen. [ms]
RTG impl. (total)	629.1	917.3
conversion	394.3	413.7
decomposition	234.8	503.6
Backend (total)	63 598.0	63 927.0

Table 2: Time spent for RTG implementation during the compilation process.

Table 2 shows the compilation time measurements for the biased coalescing strategy. This compiler configuration has the fastest register allocation and copy coalescing while producing a high number of non-trivial RTGs, which means that the relative compile time impact of our code generation scheme is larger than in all other configurations.

We divide the total time needed for implementing RTGs into the time needed for the conversion step and the time needed for the decomposition step. The two steps correspond to the conversion into permutation form and to the cycle decomposition presented in Section 5. In the default implementation of libFIRM, the conversion corresponds to the first step presented in Section 5.1 and the decomposition corresponds to the second step.

We found that the runtime of the initial conversion into permutation form is nearly identical for both systems. This is not surprising, considering that, as presented in Section 7.2, at least half of the RTGs do not require additional copies and thus can be left untouched by the conversion step. Moreover, if an RTG does require copies, on average it only requires between one and two copies, depending on the coalescing scheme. Hence, the conversion step has a low influence, both on the compile time and on the code quality.

The time needed for the decomposition step increases by a factor of 2.1. This was to be expected considering the more complex nature of our permutation instructions. To put these numbers into perspective, we included the total time spent in the backend, i.e. the total time for code selection, instruction scheduling, register allocation and emitting of assembly code. The total time spent in the backend increases by 0.5%, so the presented code generation approach does not cause significant overhead.

7.4 Code Quality

We evaluated the quality of the generated code using two experiments:

1. We performed a full run of the `CINT2000` benchmark suite, collecting precise dynamic instruction counts using our modified QEMU version.
2. We validated these results by measuring the runtime of the same executables on our FPGA prototype.

Table 3 shows the absolute number of executed instructions for each run and the change of the dynamic instruction count of the version using permutation instructions relative to the regular SPARC version. The results are shown for each of the four coalescing schemes.

As expected from the numbers presented in Section 7.2, the benefit of using permutation instructions directly depends on the quality of coalescing: the worse the coalescing, the higher the benefit of using permutation instructions. However, regardless of the coalescing scheme used, every program profited from the use of permutation instructions. For the biased coalescing scheme, suitable for just-in-time compilation scenarios, the number of executed instructions is reduced by up to 5.1%. Even using the optimal coalescing solution, permutation instructions can reduce the instruction count by up to 1.9%.

Interestingly, the use of permutation instructions can often more than compensate for a copy coalescing of lower quality: For 8 of the 11 tested programs, the executable with permutation instructions and shuffle code produced by the worst copy coalescing scheme (naive) executes *fewer* instructions than the regular SPARC version with shuffle code produced by the next best coalescing scheme (biased).

In some cases, the solution found by the ILP coalescing approach executes more instructions than the executable produced by the recoloring scheme, which can happen due to two reasons. First, if the ILP solver exceeds its timeout, the best solution found up to this point might be worse than the solution found by the recoloring scheme. Second, the cost model, which is based on statically-computed execution frequencies, might not reflect the actual runtime profile of the program. Hence, the optimal solution according to the cost model can be worse in practice.

To validate the results presented in Table 3, we measured the runtime of the same executables on our FPGA prototype. As our test platform ran at a clock speed of 80 MHz and had only 32 kB of data cache, we used the reduced input dataset distribution provided by SPEC [24]. The reduced inputs preserve the profile of the original programs while significantly reducing the runtime. We ran each executable ten times (to prevent random effects by DRAM, Linux task scheduler, interrupts etc.) and determined the lowest runtime.

Table 4 shows the runtimes of the executables. In general, the measurements on our FPGA prototype support our observations from the QEMU runs: the worse the coalescing, the higher the speedup gained using permutation instructions. Also, the magnitude of the speedups matches the magnitude of the instruction count reductions for each of the four coalescing configurations. Again, every program ran faster with permutation instructions. For the biased coalescing scheme, we measured speedups of up to 1.07×.

Besides the general inaccuracy of timing measurements, we identified two possible reasons why the numbers from Tables 3 and 4 do not exactly match with each other. First, the reduced input datasets do not perfectly preserve the program profiles. Thus, program sections that contain a lot of permutation instructions

	ILP			Recoloring			Biased			Naive		
Benchmark	SPARC	LEON3-P	Change	SPARC	LEON3-P	Change	SPARC	LEON3-P	Change	SPARC	LEON3-P	Change
164.gzip	427.3	424.5	−0.7%	428.7	424.4	−1.0%	450.5	441.8	−1.9%	542.9	454.1	−16.4%
175.vpr	2 204.9	2 199.2	−0.3%	2 209.5	2 201.8	−0.3%	2 252.9	2 229.8	−1.0%	2 309.3	2 230.7	−3.4%
176.gcc	184.5	183.7	−0.4%	184.8	183.8	−0.5%	197.1	191.8	−2.7%	215.9	191.2	−11.4%
181.mcf	64.6	63.4	−1.9%	64.7	63.4	−1.9%	68.0	66.0	−2.8%	71.5	65.9	−7.8%
186.crafty	251.4	248.9	−1.0%	251.0	249.0	−0.8%	276.1	265.3	−3.9%	315.2	267.4	−15.2%
197.parser	515.0	510.5	−0.9%	515.7	510.4	−1.0%	539.1	524.4	−2.7%	617.4	539.7	−12.6%
253.perlbmk	558.3	555.2	−0.6%	531.8	531.0	−0.1%	550.9	541.0	−1.8%	611.6	551.1	−9.9%
254.gap	243.7	243.1	−0.3%	243.6	241.3	−0.9%	257.6	252.4	−2.0%	275.4	255.9	−7.1%
255.vortex	358.9	357.0	−0.5%	361.0	358.1	−0.8%	402.1	381.6	−5.1%	467.3	396.9	−15.1%
256.bzip2	331.0	330.0	−0.3%	333.1	331.1	−0.6%	360.2	349.2	−3.1%	393.1	348.5	−11.3%
300.twolf	1 261.2	1 256.9	−0.3%	1 261.5	1 257.1	−0.3%	1 275.0	1 264.7	−0.8%	1 288.9	1 264.7	−1.9%

Table 3: Number of executed instructions (in billions) executed during a full run of the `CINT2000` benchmark suite. Results are shown separately for each coalescing scheme.

	ILP			Recoloring			Biased			Naive		
Benchmark	SPARC	LEON3-P	Change	SPARC	LEON3-P	Change	SPARC	LEON3-P	Change	SPARC	LEON3-P	Change
164.gzip	256.8	255.6	−0.5%	257.4	255.8	−0.6%	263.1	258.3	−1.8%	278.2	261.3	−6.1%
175.vpr	446.6	443.6	−0.7%	448.3	445.7	−0.6%	456.4	452.6	−0.8%	466.9	464.5	−0.5%
176.gcc	175.9	175.0	−0.5%	175.8	175.4	−0.2%	190.9	185.2	−3.0%	210.8	186.5	−11.5%
181.mcf	45.5	45.5	−0.2%	45.6	45.5	−0.2%	45.7	45.2	−1.0%	45.9	45.5	−0.7%
186.crafty	59.3	59.3	−0.1%	59.7	58.4	−2.2%	64.3	62.6	−2.8%	71.7	63.2	−11.8%
197.parser	123.7	123.6	−0.1%	126.2	123.2	−2.4%	128.5	124.7	−3.0%	139.3	125.9	−9.7%
253.perlbmk	127.9	125.2	−2.1%	124.8	123.7	−0.9%	131.6	125.0	−5.0%	141.2	126.7	−10.2%
254.gap	31.1	30.9	−0.7%	31.2	31.0	−0.4%	33.3	32.1	−3.6%	34.8	32.6	−6.3%
255.vortex	51.4	51.0	−0.7%	51.5	51.1	−0.8%	56.8	52.9	−7.0%	65.2	56.1	−14.1%
256.bzip2	171.4	170.8	−0.3%	172.2	171.2	−0.6%	177.9	175.4	−1.4%	187.3	176.5	−5.7%
300.twolf	90.9	88.7	−2.4%	91.5	89.5	−2.2%	92.6	90.6	−2.2%	95.8	91.7	−4.3%

Table 4: Runtime (in seconds) of the executables on the FPGA prototype. Reduced input data sets were used. Results are shown separately for each coalescing scheme.

	SPARC	Per RTG	LEON3-P	Per RTG	Change
ILP (best)	144 356	1.86	88 670	1.14	−38.6%
Recolor	159 511	2.04	89 274	1.14	−44.0%
Biased	534 378	2.99	275 079	1.54	−48.5%
Naive (worst)	947 439	5.12	343 582	1.85	−63.7%

Table 5: Number of instructions generated for implementing RTGs.

might be underrepresented or overrepresented in the profile of the run with the reduced input, leading to a lower or to a higher speedup, respectively. Second, the dynamic instruction counting mechanism treats all instructions equally. However, in reality, load and store instructions can take multiple cycles to execute in case of a cache miss. Frequent cache misses increase the total runtime of the program, while the time spent in the program sections that contain a lot of permutation instructions remains constant. This leads to a speedup that is lower than what one would expect from the instruction count reduction.

Table 5 shows the total number of instructions generated for implementing the RTGs of all programs of the `CINT2000` benchmark suite. The numbers confirm the expressivity of the presented permutation instructions as RTGs can be implemented more concisely, reducing the number of needed instructions by up to 63.7%. As every SPARC instruction, including our `permi` instructions, is encoded with 4 bytes, this also means that the code size induced by implementing RTGs is reduced by the same percentage. Additionally, regardless of the coalescing scheme, the average RTG can be implemented using fewer than two instructions when `permi` instructions are available, whereas up to 5.12 instructions are needed using the regular instruction set.

7.5 Permutation Reverts

We have measured the impact of permutation reversion, which is required to handle traps (see Sections 6.2 and 6.3). The measurements were performed using a performance counter in our FPGA implementation, which counts the cycles spent for reversion. Figure 13 shows the ratio of time spent for reversion compared to total application runtime. If traps were occurring with the same frequency for all applications, the ratio would be the same. However, the large spread of nearly 10^4 shows that for some applications window overflow/underflow traps (e.g. due to recursion) or traps due to I/O or syscalls occur more frequently. Still, the performance loss due to permutation reversion is always below 0.1% (i.e. 10^{-3}).

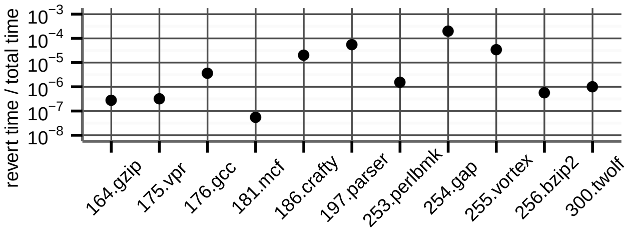


Figure 13: Ratio of time spent for permutation reversion to total runtime of each SPEC benchmark. Data gathered from FPGA prototype using the reduced input dataset.

7.6 Area Overhead

Table 6 shows the resource usage for the base system compared to the LEON3-P. The LEON3-P implementation uses multiple large multiplexers for extracting the current window and applying the new permutation to the existing one. When using an FPGA as target technology, multiplexers are realized by Look-up tables (LUTs), which explains the increased number of required LUTs. As — to the best of our knowledge — there are no publicly available memory-compilers for multi-port memories targeting ASICs, we focused our evaluation on FPGAs. However, multiplexer synthesis is discussed in [14], stating that *"Multiplexers are expensive in FPGAs and cheap in ASICs"*. Therefore, it can be assumed that the area overhead of an ASIC implementation would be considerably smaller.

	base system	LEON3-P	Overhead
LUTs	15 024 (21%)	21 630 (31%)	44%
Slices	7 249 (41%)	9 507 (55%)	31%
Flip-flops	7 607 (11%)	8 851 (12%)	16%
BlockRAMs	28 (19%)	28 (19%)	0%
Frequency	80 MHz	80 MHz	0%

Table 6: Hardware implementation comparison between base system and LEON3-P with 8 register windows. FPGA resource utilization percentage in parentheses.

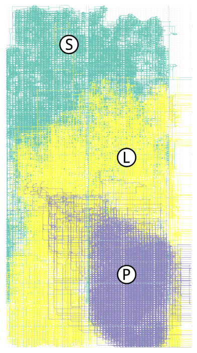


Figure 14: Floorplan of our FPGA implementation.

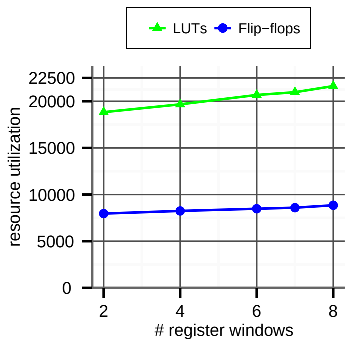


Figure 15: Design space exploration for different number of register windows.

Additional flip-flops are required for storing the logical-physical register address mapping (highlighted table component in Figures 9 and 10). Compared to the base system, there is no frequency loss, as the decode (where the extensions for register file permutation are added) and exception (where permutation reverts are performed if necessary) stages are not the critical path in the system. No additional on-chip block memory (BlockRAM) is required.

Figure 14 shows the floorplan of the placed and routed LEON3-P design on the Virtex-5 LX110T FPGA. The main logic of the permutator — multiplexers and permutation table — is in the purple area Ⓟ. The LEON 3 CPU is located in the yellow area Ⓛ, while the remaining components in the system (e.g. DDR controller, debug unit, bus arbiter, etc.) are in the green area Ⓢ.

We have synthesized the design with different amounts of register windows to analyze the impact on area. The number of LUTs and Flip-flops are shown in Figure 15. The amount of required LUTs — which contribute the largest part to the area overhead — can be reduced significantly (approximately by half) by decreasing the amount of register windows from 8 to 2. The reason is the reduction of the size of the multiplexers used for extracting the current window from the permutation table. However, programs that make use of nested function calls generally profit from a large number of register windows, thus the amount of register windows is a performance-area trade-off determined at design time.

8. Conclusion

In this paper, we presented a novel approach to accelerate the execution of shuffle code by a hardware extension for register file permutations. We demonstrated how our hardware extension can be integrated and exposed in a standard architecture. Additionally, we showed that our adapted compiler can efficiently utilize the modified hardware. The concept was evaluated using QEMU as well as an FPGA prototype executing the SPEC CINT2000 benchmark under Linux. Using our proposed extensions, the number of executed instructions is reduced by up to 5.1% while the compilation time is unaffected.

9. 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] The LLVM Compiler Infrastructure Project. http://www.llvm.org.
[2] Aeroflex Gaisler. LEON 3. http://www.gaisler.com/leonmain.html.
[3] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference (ATEC)*, pages 41–41, 2005.
[4] F. Bouchez, Q. Colombet, A. Darte, F. Rastello, and C. Guillon. Parallel Copy Motion. In *SCOPES*, pages 1–10, 2010.
[5] F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register Allocation: What does the NP-Completeness Proof of Chaitin et al. really prove? or revisiting Register Allocation: Why and How? In *LCPC*, USA, 2006.
[6] F. Bouchez, A. Darte, and F. Rastello. On the Complexity of Register Coalescing. In *CGO*, pages 102–114, 2007.
[7] F. Bouchez, A. Darte, and F. Rastello. Advanced Conservative and Optimistic Register Coalescing. In *CASES*, pages 147–156, 2008.
[8] M. Braun, S. Buchwald, and A. Zwinkau. Firm—A Graph-Based Intermediate Representation. Technical Report 35, Karlsruhe Institute of Technology, 2011.
[9] M. Braun, C. Mallon, and S. Hack. Preference-Guided Register Assignment. In *Compiler Construction*, pages 205–223, 2010.
[10] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, 1994.
[11] P. Brisk, F. Dabiri, R. Jafari, and M. Sarrafzadeh. Optimal Register Sharing for High-Level Synthesis of SSA Form Programs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(5):772–779, 2006.
[12] G. J. Chaitin. Register Allocation & Spilling via Graph Coloring. In *Compiler Construction*, pages 98–105, 1982.
[13] Q. Colombet, B. Boissinot, P. Brisk, S. Hack, and F. Rastello. Graph-Coloring and Treescan Register Allocation using Repairing. In *CASES*, pages 45–54, 2011.
[14] A. Ehliar and D. Liu. An ASIC Perspective on FPGA Optimizations. In *Field-Programmable Logic and Applications*, pages 218–223, 2009.
[15] M. J. Forsell. Are Multiport Memories Physically Feasible? *SIGARCH Computer Architecture News*, 22(4):47–54, 1994.
[16] Freescale Semiconductor. AltiVec^{TM}Technology Programming Interface Manual.
[17] D. Grund and S. Hack. A Fast Cutting-Plane Algorithm for Optimal Coalescing. In *Compiler Construction*, pages 111–115, 2007.
[18] Gurobi Optimization Inc. Gurobi Optimizer Reference Manual. http://www.gurobi.com, 2012.
[19] S. Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, 2007.
[20] S. Hack and G. Goos. Copy Coalescing by Graph Recoloring. *SIGPLAN Notices*, 43(6):227–237, 2008.
[21] S. Hack, D. Grund, and G. Goos. Register Allocation for Programs in SSA Form. In *Compiler Construction*, pages 247–262, 2006.
[22] J. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, 2000.
[23] Intel Corporation. Intel®Architecture Instruction Set Extensions Programming Reference.
[24] A. KleinOsowski and D. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1(1):7–7, 2002.
[25] M. Paleczny, C. Vick, and C. Click. The Java Hotspot^{TM} Server Compiler. In *Symposium on Java^{TM} Virtual Machine Research and Technology Symposium (JVM)*, pages 1–12, 2001.
[26] J. Park and S.-M. Moon. Optimistic Register Coalescing. *ACM Transactions on Programming Languages and Systems*, 26(4), 2004.
[27] F. M. Q. Pereira and J. Palsberg. Register Allocation Via Coloring of Chordal Graphs. In *APLAS*, pages 315–329, 2005.
[28] L. Rideau, B. P. Serpette, and X. Leroy. Tilting at Windmills with Coq: Formal Verification of a Compilation Algorithm for Parallel Moves. *Journal of Automated Reasoning*, 40(4):307–326, 2008.
[29] D. Sima. The Design Space of Register Renaming Techniques. *IEEE Micro*, 20(5):70–83, 2000.
[30] SPARC International Inc. The SPARC Architecture Manual, Version 8.
[31] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
[32] Y. Tatsumi and H. Mattausch. Fast Quadratic Increase of Multiport-Storage-Cell Area with Port Number. *Electronics Letters*, 35(25):2185–2187, 1999.