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.

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 \( x^{(R)} \) 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 \( r_0 \) 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 r₁ and r₂, just the access to these two registers is exchanged. The read access to r₁ in the last basic block will therefore lead to a read access of r₂ 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 locations) 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 $\phi$-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 adaptations 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 liveliness analysis judges the two corresponding variables to 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 $\phi$-functions are still present after register allocation [21]. The $\phi$-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 $\phi$-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 register renaming. Pereira et al. [27] 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 $\phi$-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 $\sigma$ 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.

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

(1)
Cycle notation describes a permutation \( \sigma \) 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 \( \sigma \) 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 \( d \).

\[
\sigma = \prod_i c_i
\]

\[
c_i = (x_k \sigma(x_k) \sigma^2(x_k) \ldots \sigma^{d-1}(x_k)), \forall j \neq i : x_k \notin c_j
\] (2)

Elements that are not part of any cycle are fixed points of the respective permutation, i.e., element \( x_j \) and its image \( \sigma(x_j) \) 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).

\[
c_i = (x_k \sigma(x_k) \sigma(x_k) \ldots \sigma^{d-1}(x_k)) = (x_k \ldots \sigma^{d-1}(x_k))(\sigma^d(x_k) \ldots \sigma^{d-1}(x_k))
\] (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, \( \lfloor \log_2(32) \rfloor = 5 \) 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 \( 32/\lfloor \log_2(32) \rfloor = 6 \)-element cycles, while sacrificing one register.

We have extended the SPARC V8 ISA with two instructions for permuting register files: (i) \texttt{perm15} applies a 5-cycle permutation and (ii) \texttt{perm23} 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 \texttt{perm15}, each argument corresponds to a member of the 5-cycle; For \texttt{perm23}, \( a \) and \( b \) encode 2-cycle, while the 3-cycle is encoded with \( c, d, e \). Cycles shorter than 5 are enclosed a swap of \( r2 \) and \( r3 \). \texttt{perm23} can be used to encode two 2-cycles using the same technique. The \texttt{perm15} and \texttt{perm23} instructions are discarded 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 \texttt{perm5}, otherwise as \texttt{perm23}. In the remainder of the paper we will use \texttt{perm} 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.

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 \( v_0 \in V_2 \), the start node, with no incoming edges, exactly one node \( v_{n-1} \in V_2 \), the end node, with no outgoing edges and there exists a path of length \( |V_2| \) from \( v_0 \) to \( v_{n-1} \). The component on the right consists of a subgraph that swaps the contents of registers \( r_0 \) and \( r_1 \), 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 \( n' \rightarrow n \), 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 \( r_0 \), a cycle \((r_1, \ldots, r_k)\) can be implemented by \( k + 1 \) copies following the scheme \( r_k \rightarrow r_1, r_{k-1} \rightarrow r_k, \ldots, r_2 \rightarrow r_1 \).
• 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 \(\text{xer}\) instructions, requiring \(3(k - 1)\) \text{xer} 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 \(\text{perm}\) instruction can implement a single cycle of maximum size 5, i.e. a cycle of size 2, 3, 4 or 5, while the \(\text{perm}\) 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 \(\text{perm}\) instructions is straightforward: Cycles of maximum size 5 can be directly translated into one \(\text{perm}\) instruction, while larger cycles can be decomposed into a product of cycles of maximum size 5 and then expressed using a series of \(\text{perm}\) instructions. In general, this does not generate a short instruction sequence, because it does not make use of the \(\text{perm}\) 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 \(r_3 \rightarrow r_4 \rightarrow r_5\), the middle component from Figure 4, as an example. Note that because there is no self-loop \(r_3 \rightarrow r_3\), 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 \(r_5 \rightarrow r_3\) to the graph, thereby converting it to a cycle and reducing it to the aforementioned case.

![Figure 5: Converting a chain into a cycle by adding the artificial edge](image)

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.

\[
\begin{align*}
    r_6 &\rightarrow r_7 &\rightarrow r_8 &\rightarrow r_6 &+ r_7 &\rightarrow r_6 &\rightarrow r_8
\end{align*}
\]

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.

![Figure 7: Conversion of a tree-shaped RTG into a graph in permutation form and a list of additional copies](image)

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 \(\text{perm}\) instructions as possible. To start, let us look at an example where generating optimal code is not immediately obvious.

![Figure 8: A sample RTG after conversion into permutation form](image)

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 \(\text{perm}\) instruction. The 6-cycle can be split up into a 5-cycle and a 2-cycle, thereby requiring two \(\text{perm}\) 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 c4 into a 2-cycle c2 and a 3-cycle c3 and to combine them with other small cycles using the permi23 instruction. For example, we could implement c2 and another 3-cycle c5 using a permi23 instruction, leaving us with c3. However, we now have implemented one 4-cycle using one instruction, because we started with c4 and c5 and ended up with c3, so we could have also used a permi5 for the 4-cycle. The same reasoning can be applied for c3. 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) # Rememebre 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, cycle2′))
                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.

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 $O(n)$ 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 $O(n)$ 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 $O(1)$, and thus the whole decomposition is in $O(n)$. 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.
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 perm15 instruction (similar for perm23). The permutation table is initialized with the identity (only registers r5 – r9 are shown), except for r8 and r9 which are exchanged. The process consists of four steps, all of which happen in the Decode stage:
1. The instruction decoder recognizes a perm15 or perm23 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 perm15 instruction, r5 should become r9, but as in the existing permutation in the permutation table r9 is currently in r8, r5 becomes r8. Figure 10 presents the pipeline activities of an add instruction immediately following this perm15 instruction. It shows how the existing permutation is applied to all register addresses before accessing the register file.

perm15 %r5, %r9, %r7, %r6, %r8
add %r9, %r5, %r7

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 r5 from a task (to restore it later), it will actually access the physical register where the value of r5 is currently stored (r8 in Figure 10). When restoring r5 later, it may be written to a different register, however, an access to r5 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 annuling 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.
6.3 Concept for Exception-safe Permutation

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

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

(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 $\rho$:

$$\sigma^{-1}_4 \sigma^{-1}_3 \sigma^{-1}_2 \sigma^{-1}_1(\sigma_4 \sigma_3 \sigma_2 \sigma_1(\rho)) = \rho$$

(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 four additional 25-bit wide pipeline registers and four 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.

- **Naïve** 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 and tested 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 \texttt{perm} instructions and without additional copy instructions.

<table>
<thead>
<tr>
<th>Coalescer</th>
<th>Number of RTGs</th>
<th>Average size</th>
<th>No value duplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP (best)</td>
<td>77,783</td>
<td>2.9</td>
<td>74%</td>
</tr>
<tr>
<td>Recoloring</td>
<td>78,194</td>
<td>2.9</td>
<td>74%</td>
</tr>
<tr>
<td>Biased</td>
<td>178,812</td>
<td>4.6</td>
<td>54%</td>
</tr>
<tr>
<td>Naive (worst)</td>
<td>185,035</td>
<td>6.6</td>
<td>89%</td>
</tr>
</tbody>
</table>

Table 1: Register transfer graph properties.

Table 1 shows 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.

<table>
<thead>
<tr>
<th></th>
<th>Default [ms]</th>
<th>Our code gen. [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTG impl. (total)</td>
<td>629.1</td>
<td>917.3</td>
</tr>
<tr>
<td>conversion</td>
<td>394.3</td>
<td>413.7</td>
</tr>
<tr>
<td>decomposition</td>
<td>234.8</td>
<td>503.6</td>
</tr>
<tr>
<td>Backend (total)</td>
<td>63,508.0</td>
<td>63,927.0</td>
</tr>
</tbody>
</table>

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 ×.
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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SPARC</th>
<th>LEON3-P</th>
<th>Change</th>
<th>SPARC</th>
<th>LEON3-P</th>
<th>Change</th>
<th>SPARC</th>
<th>LEON3-P</th>
<th>Change</th>
<th>SPARC</th>
<th>LEON3-P</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>256.8</td>
<td>255.6</td>
<td>−0.5%</td>
<td>257.4</td>
<td>255.8</td>
<td>−0.6%</td>
<td>263.1</td>
<td>258.3</td>
<td>−1.8%</td>
<td>278.2</td>
<td>261.3</td>
<td>−6.1%</td>
</tr>
<tr>
<td>175.vpr</td>
<td>446.6</td>
<td>443.6</td>
<td>−0.7%</td>
<td>448.3</td>
<td>445.7</td>
<td>−0.6%</td>
<td>456.4</td>
<td>452.6</td>
<td>−0.8%</td>
<td>466.9</td>
<td>464.5</td>
<td>−0.5%</td>
</tr>
<tr>
<td>176.gcc</td>
<td>175.9</td>
<td>175.0</td>
<td>−0.5%</td>
<td>175.8</td>
<td>175.4</td>
<td>−0.2%</td>
<td>190.9</td>
<td>185.2</td>
<td>−3.0%</td>
<td>210.8</td>
<td>186.5</td>
<td>−11.5%</td>
</tr>
<tr>
<td>181.mcf</td>
<td>59.3</td>
<td>59.3</td>
<td>−0.1%</td>
<td>59.7</td>
<td>58.4</td>
<td>−2.2%</td>
<td>64.3</td>
<td>62.6</td>
<td>−2.8%</td>
<td>71.7</td>
<td>63.2</td>
<td>−11.8%</td>
</tr>
<tr>
<td>186.crafty</td>
<td>123.7</td>
<td>123.6</td>
<td>−0.1%</td>
<td>126.2</td>
<td>123.2</td>
<td>−2.4%</td>
<td>128.5</td>
<td>124.7</td>
<td>−3.0%</td>
<td>139.3</td>
<td>125.9</td>
<td>−9.7%</td>
</tr>
<tr>
<td>197.parser</td>
<td>127.9</td>
<td>125.2</td>
<td>−2.1%</td>
<td>124.8</td>
<td>123.7</td>
<td>−0.9%</td>
<td>131.6</td>
<td>125.0</td>
<td>−5.0%</td>
<td>141.2</td>
<td>126.7</td>
<td>−10.2%</td>
</tr>
<tr>
<td>253.perlmbk</td>
<td>31.1</td>
<td>30.9</td>
<td>−0.7%</td>
<td>31.2</td>
<td>31.0</td>
<td>−0.4%</td>
<td>33.3</td>
<td>32.1</td>
<td>−3.6%</td>
<td>34.8</td>
<td>32.6</td>
<td>−6.3%</td>
</tr>
<tr>
<td>254.gap</td>
<td>51.4</td>
<td>51.0</td>
<td>−0.7%</td>
<td>51.5</td>
<td>51.1</td>
<td>−0.8%</td>
<td>56.8</td>
<td>52.9</td>
<td>−7.0%</td>
<td>65.2</td>
<td>56.1</td>
<td>−14.1%</td>
</tr>
<tr>
<td>255.vortex</td>
<td>171.4</td>
<td>170.8</td>
<td>−0.3%</td>
<td>172.2</td>
<td>171.2</td>
<td>−0.4%</td>
<td>177.9</td>
<td>175.4</td>
<td>−1.4%</td>
<td>187.3</td>
<td>176.5</td>
<td>−5.7%</td>
</tr>
<tr>
<td>256. bzip2</td>
<td>90.9</td>
<td>88.7</td>
<td>−2.4%</td>
<td>91.5</td>
<td>89.5</td>
<td>−2.2%</td>
<td>92.6</td>
<td>90.6</td>
<td>−2.2%</td>
<td>95.8</td>
<td>91.7</td>
<td>−4.3%</td>
</tr>
<tr>
<td>300.wolf</td>
<td>144356</td>
<td>1.86</td>
<td></td>
<td>88670</td>
<td>1.14</td>
<td>−38.6%</td>
<td></td>
<td></td>
<td></td>
<td>45.4</td>
<td>46.4</td>
<td>−0.5%</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Per</th>
<th>RTG</th>
<th>LEON3-P</th>
<th>Per</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP (best)</td>
<td>144356</td>
<td>1.86</td>
<td>88670</td>
<td>1.14</td>
<td>−38.6%</td>
</tr>
<tr>
<td>Recolor</td>
<td>159511</td>
<td>2.04</td>
<td>89274</td>
<td>1.14</td>
<td>−44.0%</td>
</tr>
<tr>
<td>Biased</td>
<td>534378</td>
<td>2.99</td>
<td>275079</td>
<td>1.54</td>
<td>−48.5%</td>
</tr>
<tr>
<td>Naive (worst)</td>
<td>947439</td>
<td>5.12</td>
<td>343582</td>
<td>1.85</td>
<td>−63.7%</td>
</tr>
</tbody>
</table>

Table 5: Number of instructions generated for implementing RTGs.

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 measurement results 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., $10^{-3}$).

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

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

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