The Journal of Supercomputing (2022) 78:12553–12588
https://doi.org/10.1007/s11227-022-04359-w

Vectorizing divergent control flow with active-lane consolidation on long-vector architectures

Wyatt Praharenka[1] · David Pankratz[1] · João P. L. De Carvalho[1] · Ehsan Amiri[2] · José Nelson Amaral[1]

Accepted: 5 February 2022 / Published online: 7 March 2022

Abstract

Control-flow divergence limits the applicability of loop vectorization, an important code-transformation that accelerates data-parallel loops. Control-flow divergence is commonly handled using an IF-conversion transformation combined with vector predication. However, the resulting vector instructions execute inefficiently with many inactive lanes. Branch-on-superword-condition-code (BOSCC) instructions are used to skip over some vector instructions, but their effectiveness decreases as vector length increases. This paper presents a novel vector permutation, Active-lane consolidation (ALC), that enables efficient execution of control-divergent loops by consolidating the active lanes of two vectors. This paper demonstrates the use of ALC with two loop transformations and applies them to kernels extracted from the SPEC CPU 2017 benchmark suite leading to up to a 30.9% reduction in dynamic instruction count compared to optimization using only BOSCCs. Motivated by ALC, this paper also proposes design changes to the ARM scalable vector extension (SVE) to improve vectorization of control-divergent loops.

Keywords Vectorization · Scalable vector extension · Control-flow divergence · Code generation · Instruction-set architecture design

✉ Wyatt Praharenka
praharen@ualberta.ca

David Pankratz
pankratz@ualberta.ca

João P. L. De Carvalho
joao.carvalho@ualberta.ca

Ehsan Amiri
ehsan.amiri@huawei.com

José Nelson Amaral
jamaral@ualberta.ca

[1] University of Alberta, Edmonton, Canada

[2] Huawei Technologies Canada, Markham, Canada

1 Introduction

Loop vectorization is an important and popular optimization to extract performance from data-parallel loops. During loop vectorization, multiple independent loop iterations are mapped to the lanes of a single vector instruction that executes quicker than performing each scalar operation individually. Interest surrounding vector architectures is growing with the introduction of vector extensions such as ARM's SVE and RISC-V "V" that bring new capabilities to accelerate a diverse collection of workloads using vector hardware. For instance, ARM's SVE has been successfully deployed in the Fugaku supercomputer by Fujitsu, that has taken the lead in the TOP500 ranking of most powerful supercomputers [1].

The challenges to vectorization presented by control flow in loops—introduced by `if-then-else` and `goto` statements—are commonly addressed by linearizing the control flow through IF-conversion [2]. IF-conversion is supported through predicated vector instructions, a common feature in modern vector ISAs such as Intel's AVX, ARM's SVE and RISC-V's "V" extension. However, predicated vector code generated after IF-conversion is often inefficient because many of the lanes in the vector may be inactive during execution and will not produce a useful result. In VL-time architectures, the presence of inactive lanes does not lower the latency of the vector instruction [3] and thus executing a vector instruction with a high ratio of active to inactive lanes is desirable. In contrast, density-time architectures [3] are designed such that the instruction latency depends on the amount of active lanes so that executing an instruction with a single active lane takes a shorter amount of time than executing the same instruction with all lanes active. However, modern vector architectures are VL-time architectures.

BOSCC instructions [4, 5] can be used to bypass execution of vector instructions containing only inactive lanes. BOSCC instructions are available in SVE and can be emulated in RISC-V "V." Figure 1 shows the vector utilization in a loop taken

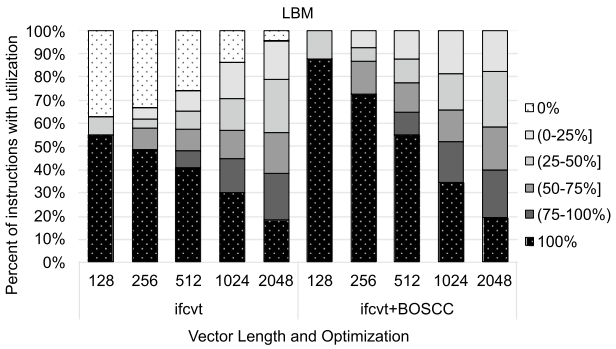


Fig. 1 Distribution of vector hardware utilization in the LBM program of the SPEC CPU 2017 benchmark suite when executing the function `StreamCollideTRT`. Darker shades indicate a higher vector utilization. This function accounts for 97% of execution time. **ifcvt:** naive predicated execution of vectorized code using flattened control-flow. **ifcvt+boscc:** the IF-converted code optimized using BOSCCs. BOSCCs detect uniform vectors to elide unnecessary execution of basic blocks. At a vector length of 2048-bits, BOSCC optimizations have nearly no effect

from the LBM benchmark, a SPEC CPU2017 benchmark, of IF-converted code before (ifcvt) and after the use of BOSCCs (ifcvt+boscc).[1] IF-conversion [2] is a compiler pass that enables vectorization of code containing control-flow by linearizing the basic blocks so that every block in the original CFG is executed and divergent control flow is handled by predicating the blocks with the corresponding condition. Each bar shows the vector utilization distribution of instructions measured at runtime. Darker bars represent higher utilization of vector registers.[2] The results indicate that BOSCC instructions are effective on short vectors and much less effective on longer vectors that are becoming increasingly common. For instance, while executed on 128-bit long vectors, BOSCCs raise the percentage of uniformly active executions from 55% to 88%; for 2048-bit long vectors, *ifcvt* and *ifcvt+BOSCC* are indistinguishable because BOSCCs inserted in the code are unable to optimize the partially active vector instructions that dominate when executing with 2048-bit long vectors. The innovations presented in this paper attempt to increase the efficiency of execution on long vector architectures in the presence of divergent control flow.

The effectiveness of BOSCC instructions decreases as vector-length grows because it is more likely that some lanes of the vector evaluate differently than the others for a given condition causing the vector to become divergent and the BOSCC instruction to fail. The penalty to the effectiveness of BOSCCs when moving to longer vectors is concerning given that the industry has been consistently increasing vector length. For instance, Intel first moved from 128-bit vectors in SSE to 256-bits with AVX and now to 512-bits in their most current vector extension, AVX-512. ARM has taken a slightly different approach, moving from 128-bit vectors in Advanced SIMD to SVE that now allows support for up to 2048-bit long scalable-vectors. This paper addresses the issues with vector utilization by improving the handling of control-flow divergence, especially in long-vector architectures.

This paper proposes a new permutation, Active-Lane Consolidation (ALC), to improve vector utilization in the presence of divergent control flow where state-of-the-art optimizations fall short. The goal of the ALC permutation is to consolidate active lanes from two partially active vectors into a single vector register with the aim of forming uniform vectors that BOSCC instructions are able to optimize. We propose two loop-transformations that use the ALC permutation to consolidate vectors corresponding to a conditional block in the control-flow graph in attempt to increase vector utilization. The unrolling ALC transformation consolidates instances of vector instructions in a vectorized loop from two consecutive iterations exposed by loop-unrolling. The second transformation, iterative ALC, broadens the range that lanes can be consolidated from to several iterations by decoupling the consolidated vector from the loop, allowing for active lanes in any iteration during the loop to be moved into the consolidated vector.

This paper makes the following contributions:

[1] For details of the experiment refer to Sect. 6.1.

[2] In this paper, vector registers are referred to simply as *vectors*.

- Active-lane consolidation, a new permutation operation that consolidates active lanes to increase vector utilization. ALC includes a mechanism to also retain the inactive lanes, allowing ALC to be applied to arbitrary control flow (Sect. 3.1).
- An implementation of the ALC permutation to demonstrate that ALC readily maps to existing vector architectures.
- Two loop transformations, Loop-Unrolling ALC (Sect. 3.2) and Iterative ALC (Sect. 5), that marry ALC and BOSCCs to improve vector utilization leading to a reduction in the number of instructions executed.
- A proposal for a new vector permutation instruction, motivated by ALC 's efficacy, that would provide out-of-the-box lane consolidation (Sect. 4.4).
- Emulation-based case studies that show the applicability of the ALC permutation and the proposed loop-transformations to the SPEC CPU 2017 benchmark suite and the resulting benefits (Sect. 6).

2 Background

SIMD execution and vector extensions Often, Single-Instruction stream Multiple-Data stream (SIMD) architectures [6] operate on vector registers. A vector register is organized into *vector lanes* with each lane able to hold a single vector element. CPUs support SIMD execution through vector ISA extensions—e.g., Intel AVX [7] and ARM's Advanced SIMD [8]—that define arithmetic and logic vector instructions for a number of data types. Vector operations may operate between elements in the same lane of the operand vectors or across lanes within a single vector.

The size of the vector registers or *vector length (VL)* can vary greatly between architectures. For instance, Intel has three popular vector extensions: AVX (128 bits), AVX2 (256 bits), and AVX512 (512 bits) each defining instructions operating on varying vector lengths. Each vector register can hold data types with different bit widths. Thus, the number of lanes in a vector register with a certain `VL` varies with the data type. For example, if a 128-bit vector register holds 64-bit types it will have two lanes. In this paper, the number of lanes in a vector register is called the *vector element count* (`EC`).

According to Stephens et al. there is no single best vector length [9]. Logic design complexity, power consumption, chip area, and the target application domains influence the choice of `VL`. For example, the CRAY-1 [10] supercomputer featured 4096-bit vector registers while a typical low-power Intel processor with the Streaming-SIMD-Extension (SSE) has 128-bit vector registers. For years, the length of vectors has been coupled to the instruction encodings leading to an unnecessary proliferation of instructions when architectures expand the vector length [11]. To prevent this proliferation while addressing the question of what vector length to use, ARM's vector-length-agnostic (**VLA**) Scalable-vector extension (SVE) [12] decouples the `VL` from the vector instruction encoding and delegates the choice of `VL` to the hardware implementation. Another ISA choosing to follow the emerging `VLA` architecture is the RISC-V "V" vector extension [13].

Listing 1: A simple data-parallel loop with control flow

```
1 double * a, * b, * c, * d;
2
3 for (int i = 0; i < 1000; i++) {
4   if (a[i] < b[i])
5 B1:   c[i] = (a[i]*a[i] + b[i]*b[i]) / (d[i]*d[i]);
6   else
7 B2:   c[i] = (a[i] - b[i]) / d[i*2];
8 }
```

Listing 2: SVE assembly of the vectorized loop in Listing 1

```
1  // x0: i = 0 x1: d x2: c x3: b x4: a w3: 1000
2
3  whilelo p2.d, xzr, w3
4  .L1:
5    ld1d z1.d, p2/z, [x3, x0, lsl #3]
6    ld1d z0.d, p2/z, [x4, x0, lsl #3]
7    fcmlt p0.d, p2/z, z0.d, z1.d
8    bic p1.b, p3/z, p3.b, p0.b
9    movprfx z2, z0
10   fmul z2.d, p0/m, z2.d, z0.d
11   fmla z2.d, p0/m, z1.d, z1.d
12   index z4.d, x0, #2
13   ld1d z3.d, p1/z, [x1, z4.d, lsl #3]
14   fsub z0.d, p1/m, z0.d, z1.d
15   ld1d z1.d, p0/z, [x1, x0, lsl #3]
16   fdiv z0.d, p1/m, z0.d, z3.d
17   fmul z1.d, p0/m, z1.d, z1.d
18   movprfx z0.d, p0/m, z2.d
19   fdiv z0.d, p0/m, z0.d, z1.d
20   st1d z0.d, p2, [x2, x0, lsl #3]
21   incd x0
22   whilelo p2.d, w0, w3
23   b.any .L1
```

Loop vectorization Hand-writing SIMD code fine-tunes performance but is a tedious and error-prone task. Vector intrinsics, exposed in a programming language, offer a compromise between productivity and performance. However, the most productive approach for the generation of vector code relies on a compiler that transforms sequential scalar code into vector code through a process called *automatic vectorization* or *SIMDization* [14]. Often, the majority of the runtime of an application is spent in data-parallel loops, as such, loops are a primary target for vectorization. Modern compilers implement *loop-vectorizers* that *widens* each scalar operation appearing in a loop into a vector operation where consecutive iterations of the scalar loop map to lanes of a vector. To ensure correctness, an extensive list of legality checks is performed prior to loop vectorization. One important check uses memory dependence analysis [15, 16] to ensure that either there are no loop-carried dependencies between memory accesses in the loop or that the dependence distance

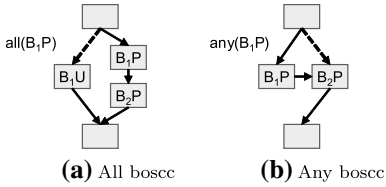
is greater than the number of elements in each vector. Control structures, such as an `if-then-else` statement, introduce control dependencies that must also be addressed by the vectorizer as discussed below.

Listing 1 shows an example of a vectorizable loop written in C. The vectorized version of this example, in SVE assembly, is shown in Listing 2. Assuming that a `double` data type contains 64 bits and the architecture implements vector registers that are 512-bits wide, each vector register will have eight lanes (`EC = 8`). The comment on line 1 specifies the initial values in the corresponding scalar registers. After vectorization, lines 5 and 7 from Listing 1 are translated into various predicated vector arithmetic instructions. For example, the fused-multiply-add vector instruction on line 11 of Listing 2 will each perform eight multiply-add operations that correspond to iterations `i` to `i+7` of the scalar loop. The governing predicate `p` register in each instruction contains Boolean values to indicate which lanes are active. The values in the `p` registers are generated by the `fcmlt` and `bic` instructions that compute the predicate for the *if* and the *else* block in Listing 1.

While contiguous memory accesses are well-supported by most vector extensions and trivial to generate vectorized code for, non-contiguous accesses are not. For instance, the load `d[i*2]` on line 7 has a stride of two with consecutive lanes of this load accessing memory addresses: $\{d+0, d+2, \ldots, d+2*(EC-1)\}$. SVE is equipped with a gather load instruction, shown on line 13, that enables the vectorization of the load `d[i*2]`. In ISAs that do not support gather and scatter instructions, these non-contiguous accesses can be accomplished by scalarizing the memory access but such a solution is much slower than a tailor-made gather/scatter instruction.

Vectorizing control dependencies Control dependencies are introduced by conditional statements and must be handled properly when generating vectorized code. For example, in Listing 1 blocks B_1 and B_2 are control dependent on the block that computes the condition `a[i] < b[i]` (line 4). This control dependency is loop variant because the condition depends on the value of the loop induction variable `i`. A loop-variant dependency is a *divergent condition*—such conditions are also referred as *varying conditions* [17]—while a loop-invariant control-flow dependency, where the condition does not depend on the loop induction variable, is called a *uniform condition*. Uniform conditions can often be hoisted out of the loop by loop unswitching [18].

IF-conversion [2]—also referred to as *control-flow linearization* [17, 19]—is a technique to enable the vectorization of loops containing control flow. *IF-conversion* is applied to the control-flow graph of a loop prior to loop vectorization to convert control dependencies into data dependencies. IF-conversion replaces a branch statement with the computation of predicates, one for each successor of the branch and assigns these predicates accordingly to each successor block in the IF-converted code. As the branch in the original code is removed, all successor blocks will execute at runtime and in this aspect, the blocks are linearized. The predicated statements replace the behavior of the branch to control execution as only predicated statements whose predicate evaluates to true at runtime produce a result, thus the control dependence that once existed due to the presence of a branch now exists as a data dependence on the variable holding the predicate.



(a) All boscc **(b)** Any boscc

Fig. 2 BOSCCs used to optimize the IF-converted code

A vector is *uniform* either when all of the lanes are active or all of the lanes are false and is *divergent* when some of the lanes are active while others are not. The IF-converted and vectorized code for Listing 1 is shown in Listing 2 where all branches are removed and replaced by instructions to produce vector predicates. Line 7 shows the `fcmlt` instruction to generate the predicate for B_1 and the `bic` instruction on line 8 that generates the predicate for B_2. Operations inside blocks B_1 and B_2 are vectorized and predicated by these predicates.

In ISAs that do not support predicated execution the control dependencies can be handled via IF-selection [20] or scalarization. Vector `select` instructions perform an element permutation that retrieves values from the first input vector if the lane is active or from the second otherwise. Scalarization performs scalar operations on each lane of the vector iteratively and may require expensive packing and unpacking of the lanes of the vector register into scalar registers.

Branches on super-word condition codes Predicated vector instructions selectively enable and disable lanes, with disabled lanes producing no result. The ratio of active lanes to total lanes in a vector is called the *vector utilization*. Lower utilization leads vector instructions to consume CPU resources while producing few or no useful results. In addition, most modern vector architectures feature VL-time performance rather than density-time performance where the number of inactive lanes present in a vector does not affect the latency of the predicated instruction [3]. Inactive lanes in VL-time architectures are wasted opportunity and thus, it is desirable to have code with high vector utilization.

BOSCC [4, 5] instructions can be used to elide execution of vector instructions that have no active lanes. A BOSCC instruction is a branch instruction that branches based on whether a super-word (vector) satisfies a certain condition. For instance, SVEs condition codes `NZCV` are set as the result of vector compare instructions, e.g., `cmpeq`. Vector compares clear the flag `Z` if any lane of the resulting predicate becomes active. A `b.none` BOSCC in SVE, as shown in Fig. 3, branches when the flag `Z` is set, only taking the branch if at least one active element exists in the predicate.

Two common BOSCCs are the *any* BOSCC that branches when any lane of the vector is active and the *all* BOSCC that branches only if all lanes are active. Figure 2 shows how *any* and *all* BOSCCs can be used to optimize vectorized code for Listing 1. A dashed line indicates the edge a BOSCC branch introduces into the control flow graph. Figure 2a inserts an *all* BOSCC in a manner described in WCCV [21] to branch to block B_1U if every lane is active for the B_1 block. Since B_1U is only executed if all lanes are active, this block can contain unpredicated vector code. Figure 2b uses an *any* BOSCC instruction to skip the execution of a predicated block

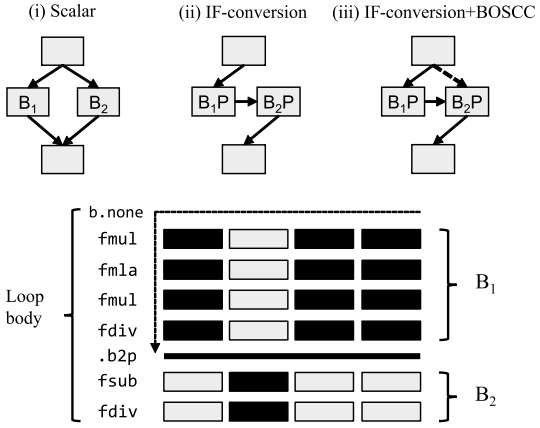


Fig. 3 Example vector execution of Listing 1 with an *any* BOSCC inserted to skip B_1

when no lanes are active; this method of utilizing an *any* BOSCC is described in Partial Control Flow Linearization [17]. The *all* BOSCC instruction improves lane utilization in the presence of biased branches while *any* BOSCC instruction bypasses IF-converted code in blocks that rarely execute with active lanes [21].

Figure 3 shows one example of a possible execution of the control flow graph in Fig. 2b once vectorized. This example execution illustrates how BOSCCs can break down in the presence of divergent vectors. In the example, the predicate for B_1P contains only a single active lane, as the BOSCC condition is true when *none* of the lanes are active, the branch to skip B_1P is not taken and executes the vector code with only a single active lane (25% utilization). Increasing vector length reduces the effectiveness of BOSCC instructions as divergent vectors may become more likely.

3 Optimizing divergent vectors with ALC

Figure 1 shows that using BOSCCs after vectorization and IF-conversion works well to optimize execution on short vectors as they are likely uniform because of the small amount of lanes they hold. The figure also indicates that the benefit of optimization with BOSCCs decreases as the vector size increases with the improvement becoming negligible at a vector length of 2048 bits. Thus, architectures that implement modern vector ISAs, which are supporting increasingly longer vectors, require new compilation approaches. In the effort towards this goal, this section introduces Active-Lane Consolidation (ALC), a vector permutation that increases vector uniformity to create more opportunities for profitable deployment of BOSCC instructions. In addition, Sect. 3.2 presents loop-unrolling ALC, a loop transformation in which ALC can be used to improve performance.

An application of ALC to a hot loop from the NAB (Nucleic Acid Builder) benchmark—a molecular-dynamics application that is part of the SPEC CPU 2017 suite—illustrates how this transformation can lead to performance improvements. Listing 3 shows a simplified excerpt of one of NAB's hot loops where, for each iteration of the loop, only one of the blocks, $B_1 - B_5$, in the `if-else-if`

chain is executed. No loop-carried memory dependencies exist within this loop and thus it can be vectorized. Profiling the run-time branch behavior reveals that B_2 is executed in approximately 85% of the iterations. However, iterations that do not execute block B_2 are interspersed with iterations that execute B_2, leading to divergent vectors when this loop is vectorized. To improve the performance of this vectorized loop, ALC consolidates active lanes from divergent vectors in consecutive iterations into uniform vectors.

Listing 3: Simplified loop from NAB

```
1  for (k = 0; k < lpears[i]+upears[i]; k++) {
2    ...
3    r2 = ...
4    dij1i = ...
5    dij = r2 * dij1i;
6    if (dij > rgbmax - sj) {            // C1
7      B1:
8      sumi -= ...
9    } else if (dij > 4.0 * sj) {        // C2
10     B2:
11     dij2i = dij1i*dij1i
12     ...
13     sumi -= ...
14   } else if (dij > ri + sj) {         // C3
15     B3:
16     sumi -= ...
17   } else if (dij > fabs(ri - sj)) { // C4
18     B4:
19     sumi -= ...
20   } else if (ri < sj) {               // C5
21     B5:
22     sumi -= ...
23   }
24 }
```

The runtime bias toward condition C_2 implies that in many iterations of the vectorized loop, the linearized basic blocks B_1, B_3, B_4, B_5 in the IF-converted code are unnecessarily executed. One solution is to unroll the vectorized loop and then to use ALC to consolidate the active lanes of divergent vectors into a single vector but this approach is limited to consolidating active lanes only from two consecutive iterations. The other solution, iterative ALC, decouples the consolidating vector from the loop iteration allowing active lanes from any iteration during the loop to be consolidated. In both schemes, the consolidated vector is stealing active lanes from future iterations of the vectorized loop in an attempt to form a uniform vector. If ALC is able to re-organize the lanes of two divergent vectors into a uniform vector, then BOSCCs can ensure that the vector loop executes only the necessary conditional block for this uniform vector. In the NAB example, ALC finds new opportunities to skip unnecessary executions of basic blocks B_1, B_3, B_4, and B_5 by consolidating the lanes in divergent vectors from subsequent iterations of the vectorized loop that execute block B_2.

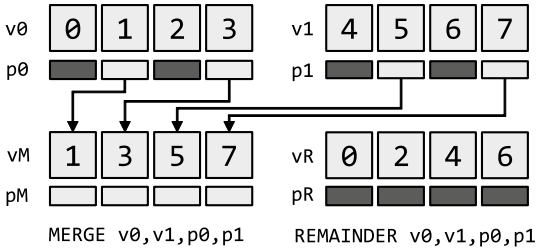


Fig. 4 The ALC permutation performed on two divergent vectors. Light colored squares represent active lanes while dark colored lanes inactive

Based on the predicates generated by IF-conversion, ALC moves as many active-lanes of two vectors as possible into a single vector and fills a second vector with the remaining inactive lanes to handle the case when those inactive lanes follow a different path of control. Once the ALC permutation is applied on a pair of vectors, the order of the lanes will have changed and for correctness, all vectors that are used as an operand inside a consolidated block must be permuted in the same fashion. This section introduces the *indexed-based inter-register permutation* to solve this in light of the large number of instructions required by ALC in actual implementation. The ALC permutation is complex in comparison to existing vector permutations so that an implementation using only current instructions may be expensive depending on the capability of the chosen ISA. This section details a proposal for a new vector permutation using SVE as the base ISA that would solve some of the shortcomings of implementing ALC using only existing instructions.

3.1 Active-lane consolidation permutation

Figure 4 illustrates the application of ALC to two vector registers based on their predicate values. In this example, the vectors `v0` and `v1` contain the indices of the loop (created with an `index` instruction) and the predicate vectors `p0` and `p1` indicate the lanes that are active (light gray) or inactive (dark gray). ALC attempts to fill the merged vector (`vM`) with active lanes spread between the two input vectors. The remaining lanes, active or not, are placed into the remainder vector (`vR`). Let `AL` be the number of active lanes. If $\texttt{AL} \geq \texttt{VL}$, then `vM` is uniform with all lanes active. If $\texttt{AL} \leq \texttt{VL}$, then `vR` is uniform with all lanes inactive.

If `vM` is a uniformly active vector with respect to a condition C_2 after consolidation then BOSCCs will execute only B_2. Given that the conditions of the `if-else-if` statement are mutually exclusive, the consolidated vector will have an all-false predicate for conditions other than C_2 and thus BOSCCs can bypass execution of the blocks these conditions guard. In general, there are no guarantees about the uniformity of the remainder vector `vR` and it may have to be processed through the entire linearized graph. However, an additional BOSCC can be inserted to check if `vR` is all-false for C_2 to further optimize execution.

Figure 4 presents ALC as a permutation of the two vectors `v0` and `v1` to create two new vectors `vM` and `vR`. Abstractly, ALC can be viewed as a compaction on a vector created by concatenating `v0` and `v1`. The AVX512 `compress` and the SVE

`compact` instructions already perform the compaction but only on a single vector register and do not retain inactive lanes in the result.

Similar to the `compact` permutation, the ALC permutation retains the relative order of the active lanes. However, due to complexities encountered in the implementation for SVE, this is not the case for inactive lanes. The lack of relative order in the inactive lanes did not have an impact in our experiments but may become important later when new features are added to vector ISAs, for example, instructions to detect intra-vector dependencies [22].

3.2 Loop-unrolling ALC

The goal of ALC is to consolidate lanes from multiple vectors to create a uniform vector. Thus, multiple vectors must be available for this consolidation. Unrolling the vectorized and IF-converted loop by a factor of two yields two vectors that can be consolidated by the ALC permutation into two new vectors, one containing mostly active lanes and the other containing mostly inactive lanes. This transformation is called "Unrolling ALC."

Figure 5a shows the control flow graph of the scalar loop from the NAB kernel in Listing 3. The conditions C_1 to C_5 control the execution of blocks B_1 to B_5. Two transformations are applied to the code from Fig. 5a to Fig. 5b: IF-conversion and vectorization. Vectorization widens each scalar instruction in Fig. 5a so that each vector instruction in Fig. 5b performs the operation for `EC` iterations of the scalar loop. The letter `P` after the block name in Fig. 5b indicates that the block is predicated by a Boolean expression composed of the conditions that control each block's execution. For instance, the condition for the predicated block B_2P in Fig. 5b is $C_2 \wedge \neg C_1$ because B_2 executes when C_2 is true (Listing 3, Line 9) and C_1 is false (Listing 3, Line 6). From Fig. 5b–c the linearized and vectorized loop is unrolled by two so that each iteration of the new loop contains two copies of the body of the loop in Fig. 5b. Blocks to process the second iteration are annotated with I_2 in Fig. 5c.

The most executed block in this loop, B_2, is the target for consolidation. In Fig. 5d the active lanes of the vectors from the first and second iterations of the unrolled loop are consolidated by ALC based on the predicates for B_2P and the other in B_2I_2P. Let P_2I be the predicate of B_2P and P_2I_2 be the predicate of B_2I_2P, then, after ALC, `vM` is a uniform vector if and only if $popcnt(P_2I) + popcnt(P_2I_2) \geq VL$. This condition is checked by counting the number of active lanes using population-count instructions on the pair of predicate registers corresponding to the condition that is being consolidated. If this condition in Fig. 5d is true, then the code can branch to the optimized path, labeled `p_opt`. Otherwise, the unoptimized path that executes the if-converted code `p_unroll` is taken. The `p_opt` path bypasses execution of B_1P, B_3P, B_4P and B_5P for the merged vector `vM` because it is uniform with respect to C_2. This is reflected in Fig. 5d as B_2U is the only block that processes `vM`. The `p_opt` path must still conservatively process `vR` through the fully if-converted path represented by blocks B_1RP to B_5RP for correct execution because

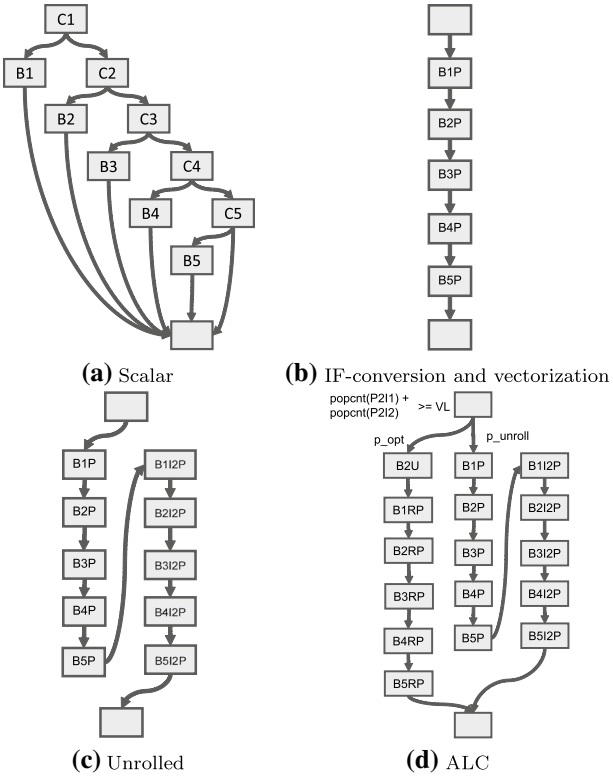


Fig. 5 Control flow graphs after: **a** Scalar, **b** IF-conversion and vectorization of **a**, **c** Unrolling of **(b)**, **(d)** ALC on **(c)**

vR is not guaranteed to be uniform. In some instances, vR may have no lanes active for this predicate and thus, inserting an *any* BOSCC to skip the block B_2RP, may be worthwhile.

The idea of consolidating active lanes is applicable to more than a pair of vectors; ALC could unroll the loop by a factor greater than two to expose three, four, or more vectors for consolidation. Given the limitations of permutation instructions currently in SVE, this paper only considers unrolling by a factor of two and consolidating the resulting pair of vectors. Unrolling the loop further and consolidating more than a pair of vectors would add additional complexity into the ALC permutation.

3.3 Tracking lane indices through permutation

Often, several vector operands within a single conditional block must be consolidated in the same fashion. This is necessary in order to ensure that the lanes of vectors defined outside of the block being consolidated are in the correct consolidated order when used within the consolidated block. Listing 4 illustrates this issue: after vectorization, the order of the lanes of variable op matches the unit-stride of the loop induction

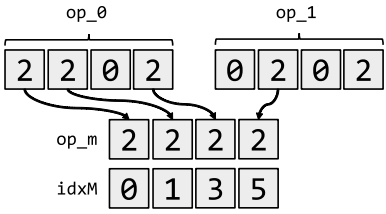


Fig. 6 Inter-vector permutation to create the merged operand by permuting the concatenated vectors of the unrolled iterations based on the consolidated index vector `idxM`

variable `i` with lane 0 holding the value for iteration `i`, lane 1 holding the value for iteration `i+1` and so forth. After the ALC permutation is performed, the variable `op` must also be permuted so the lanes match the order resulting from ALC.

In some vector extensions, such as in SVE, the implementation of the ALC permutation shown in Fig. 4 is expensive and therefore a more efficient solution consists of performing ALC on a pair of vectors holding the indices 0 to VL-1 and VL to 2*VL-1 and using the result in an index-based inter-register permutation. For instance, in Fig. 4 `EC` = 4 and the vectors `v0` and `v1` hold the indices in the original order that the induction variable is incremented. After ALC, the consolidated index vectors are `vM` and `vR`, which are used to permute all other operands used in the conditional block. The indices in `vM` and `vR` are also used to compute the addresses of gather loads and scatter stores appearing inside the consolidated block.

3.4 Permuting instruction operands

Two issues deserve further examination: 1. a consolidated block may use an operand that was defined outside of the block; and 2. a load inside a consolidated block may depend on the loop induction variable. For instance, Listing 4 shows a scalar loop while Listing 5 shows the loop after vectorization, unrolling, and ALC is applied on the `if` condition. The variable `op` is defined outside of the conditional block B_1 being consolidated; and the load `C[i]` depends on the loop induction variable `i`.

The notation `(pred) expr;` in lines 28-26 of Listing 5 represents predicated execution where `pred` is the predicate and `expr` is the expression executed if `pred` evaluates to true. The annotation `_0` or `_1` indicates to which of the two iterations exposed by unrolling each variable belongs. The original consecutive index vectors are created in lines 2 and 3. Line 19 performs ALC based on the predicates generated by the condition in line 4 of Listing 4 to create the new index vectors `idxM` and `idxR`. These indices are used to permute vectors `op_1` and `op_2`, in lines 22 and 26 before their use in the consolidated block. Figure 6 illustrates the index-based inter-vector permutation that appears in line 22 in Listing 5 within the consolidated block (B_1).

Listing 4: Example loop where the op variable must be permuted and contiguous loads converted to gather loads to correct the order to the consolidated iterations if ALC is applied.

```
1 for (int i = 0; i < N; i++) {
2   op = A[i] * 2
3
4   if (op) {              // Consolidate
5     out += op * C[i]     // B1
6   } else {
7     out += op / D[i]     // B2
8   }
9 }
```

Listing 5: Example loop from Listing 4 with vectorization,unrolling and ALC applied. Operands are merged accordingly and contiguous loads inside the consolidate block are converted to gather loads.

```
1  for (int i = 0; i < N; i += 2*VL) {
2    idx_0 = index(i,i+VL,1)   ;       // 0,1,2,3
3    idx_1 = index(i+VL,i+2*VL,1);     // 4,5,6,7
4
5    a_0 = vld(A[i]) ;         // 1,1,0,1
6    a_1 = vld(A[i+VL]);       // 0,1,0,1
7
8    op_0 = a_0*2;             // 2,2,0,2
9    op_1 = a_1*2;             // 0,2,0,2
10
11   cond_0 = op_0 != 0;       // 1,1,0,1
12   cond_1 = op_1 != 0;       // 0,1,0,1
13
14   // Check whether ALC is beneficial, 3+2 >= 4
15   if (popcnt(cond_0) + popcnt(cond_1) >= VL) {
16
17     // ALC applied to index vectors
18     // idxM = 0,1,3,5;  idxR = 2,7,4,6
19     idxM,idxR = alc(cond_0,cond_1,idx_0,idx_1,);
20
21     // Process merged vector, no need to execute code for B2
22     op_m = permute(op_0,op_1,idxM);     // 2,2,2,2
23     out += op_m * gather(C+i,idxM);
24
25     // Process remainder vector, IF-conversion
26     op_r = permute(op_0,op_1,idxR);     // 0,2,0,0
27     cond_r = op_r != 0;          // 0,1,0,0
28     (cond_r)  out += op_r * gather(C+i,idxR);
29     (!cond_r) out += op_r / gather(D+i,idxR);
30   }
31   else {
32     // Fallback execution, IF-conversion
33     (cond_0)  out += op_0 * C[i];
34     (!cond_0) out += op_0 / D[i];
35     (cond_1)  out += op_1 * C[i+VL];
36     (!cond_1) out += op_1 / D[i+VL];
37   }
38 }
```

Load/store operations that are inside a consolidated block are converted to gather/scatter operations indexed by the consolidated induction vectors. For instance, ALC requires that the contiguous load `C[i]` in Listing 4 be converted to the gather loads indexed by `idxM` or by `idxR` in lines 23 and 28 of Listing 5. Gather and scatter accesses tend to be slower than contiguous accesses because they issue more micro-operations and may access a larger number of cache lines.

4 Active-lane consolidation in SVE

To demonstrate the efficacy of ALC in mitigating divergence we develop an implementation using ARM's SVE [12], a widely adopted extension with supporting tools for code generation and emulation. The permutation primitives and transformations presented in Sect. 3 are also applicable to other long-vector architectures such as RISCV "V." This section details how the ALC primitives map to SVE instructions and presents a brief overview of the instructions' semantics. An analogous mapping could be developed to apply ALC to other vector ISAs. This section concludes by outlining challenges discovered in mapping ALC to SVE and presenting additions to SVE that help mitigate these challenges.

4.1 SVE permutations

The following SVE instructions are illustrated with examples in Fig. 7 where a light-grey predicate is true and a dark-grey predicate is false.

Select For each true predicate, take the elements from the first input vector register and for each false predicate take the elements from the second input vector register.

Splice The predicate register defines a range from the first true predicate to the last true predicate. Elements within this range are taken from the first vector and placed at the start of the result register. The remainder of the result register is filled with elements from the second vector register starting from the element in the lowest position.

Tbl In this programmable table lookup illustrated in Fig. 7c, `z0` is the data register and `z1` holds indices into `z0`. The result register receives elements from the data register based on the values in the index register. Lanes where the index register has a value outside of the interval $[0, \text{VLEN} - 1]$, where VLEN is the vector length, lead to the result register receiving the zero value. For instance, in Fig. 7c the index 4 is outside of the [0,3] range.

Compact Active elements of the input vector register are placed compactly at the start of the output register. The remaining elements in the output register are zeroed.

Index Creates a vector of incrementing values. Two scalar values, supplied either in registers or as immediate values, specify the start value and the incrementing step.

Whilelt Given two scalar registers `x0` and `x1`, generates a predicate register where the first `x1-x0` lanes are active and all other lanes are inactive.

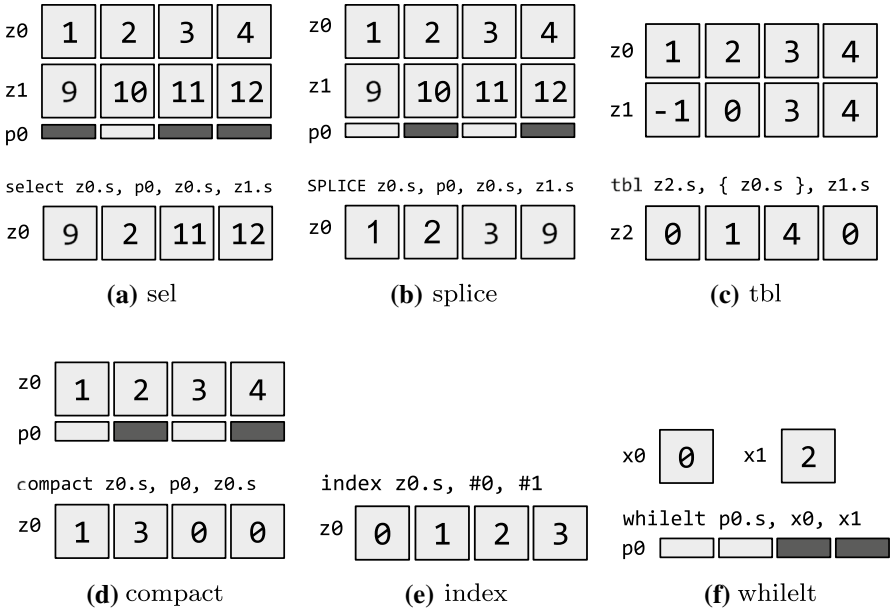


Fig. 7 Semantics of SVE permutation instructions and index/whilelt instructions used to implement the active-lane consolidation and multi-register permutation

The name `whilelt` highlights that the value of the predicate for each lane is set by testing the condition `x0` < `x1` and then incrementing `x0` before testing for the next lane.

4.2 Active-Lane Consolidation in SVE

Listing 6 shows the implementation of the ALC permutation in SVE. The variables in lines 2–6 are the input to the permutation: the original loop indices `z0` and `z1`; the predicate registers `p0` and `p1`; and the governing predicate `p2`. The governing predicate controls the execution of the loop-trip and is all true except for the tail-end of the loop when the trip count is not a multiple of `VL` or if the loop has a data-dependent exit that is taken before the loop terminates by reaching the loop bound. The comments on each line provide an example value for the initialization of these variables and later show the value that results from the execution of each instruction. In this example, the value of the predicates `p0` and `p1` in lines 4 and 5 correspond to the values of `cond_0` and `cond_1` in lines 11 and 12 of Listing 4. The SVE population count instruction, `cntp` in shown on line 14 and counts the number of active lanes in a predicate. The `.s` in the instruction indicates the vector element width which is 32 bits in this example.

Listing 6: Implementation of the ALC permutation in SVE described in Section 3.1. Example values show in this listing are consistent with the prior examples.

```
1  // EC=4, element count, i.e. VL/DTYPE_SIZE
2  z0: first vector containing indices  // = 0 1 2 3
3  z1: second vector containing indices // = 4 5 6 7
4  p0: predicate for z0 // = 1 1 0 1
5  p1: predicate for z1 // = 0 1 0 1
6  p2: governing predicate
7
8  compact z2.s, p0, z0.s           // = 0 1 3 0
9
10 compact z3.s, p1, z1.s           // = 5 7 0 0
11
12 not p3.B, p2/z, p0.B             // = 0 0 1 0
13 compact z4.s, p3, z0.s           // = 2 0 0 0
14 cntp x2, p3, p3.s                // = 1
15
16 not p4.B, p2/z, p1.B             // = 1 0 1 0
17 compact z5.s, p4, z1.s           // = 4 6 0 0
18
19 cntp x0, p1, p1.s                // = 2
20 whilelt p4.s, xzr, x0            // = 1 1 0 0
21 splice z3.s, p4, z3.s, z5.s      // = 5 7 4 6
22
23 cntp x1, p0, p0.s                // = 3
24 whilelt p3.s, xzr, x1            // = 1 1 1 0
25
26 whilelt p4.s, xzr, x2            // = 1 0 0 0
27
28 // Merge
29 splice z2.s, p3, z2.s, z4.s      // = 0 1 3 5 - all true
30 // Remainder
31 sel z4.s, p4, z4.s, z3.s         // = 2 7 4 6
```

The SVE ALC permutation shown in Listing 6, and illustrated in Fig. 8, is used to create the consolidated loop induction vectors that are used both for the index-based inter-register permutation of all the operands in the block and for the gather-load and scatter-store operations. After this sequence the merged vector, `z2.s`, contains up to `VLEN` elements from both `z0` and `z1` corresponding to the active lanes in `p0` and `p1` while the remainder vector, `z3`, contains all other elements in `z0` and `z1` that were not consolidated into `z2`. If the sum of the number of active lanes between `p0` and `p1` is greater than the vector length, then `z2` is guaranteed to be uniform with respect to the condition that created the predicates `p0` and `p1`.

4.3 Inter-register indexed permutation in SVE

In SVE an index-based inter-register permutation can be implemented using a chain of `tbl` instructions as shown in Listing 7. The `tbl` on line 7 selects elements from the first vector data register that corresponds to the first iteration in the unrolled

loop. The last lane receives a zero value because the index 5 is out of range for `VL = 4`. Line 10 normalizes the indices to the second vector data register by subtracting the element count from all lanes. Line 11 gathers the values from the second data vector with another `tbl` and the first three lanes take a zero value again because −4, −3 and −1 are outside the range of valid indices. Finally, adding the result of the two `tbl` instructions gives the final merged operand in the order created by performing the ALC permutation.

Listing 7: SVE-specific pseudo-code for the inter-register permute as introduced in Section 3.4.

```
1  // EC=4, element count, i.e. VL/DTYPE_SIZE
2  z0 = first data vector              // = 2,2,0,2
3  z1 = second data vector             // = 0,2,0,2
4  z3 = merged indices                 // = 0,1,3,5
5
6  // Data from first vector
7  tbl z4.s, { z0.s }, z3.s            // = 2 2 2 0
8
9  // Data from second vector with adjusted index
10 decw z3.s                           // = -4 -3 -1 1
11 tbl z5.s, { z1.s }, z3.s            // = 0 0 0 2
12
13 // Combine data from TBLs
14 ptrue p0.s
15 add z4.s, p0, z4.s, z5.s            // = 2 2 2 2
```

The inter-register indexed permute makes extensive use of the `tbl` instruction. ARM's previous vector extension, Advanced SIMD, featured a `tbl` instruction that took a register-table as input i.e., two or three consecutive vector registers. However, the `tbl` in the SVE specification does not include this feature because the SVE designers considered them not naturally vector-length agnostic [12]. The proposed inter-register indexed permute presents a case for the inclusion of a multi-register-table permutation in SVE because indices into the vector table can be generated by other VLA instructions such as `index`.

4.4 Proposal for native support in SVE

There are two shortcomings in the SVE design that introduce inefficiencies when implementing ALC: there are no native instructions that directly map to the ALC permutation, and there is no support for indexed permutations on multi-register tables. These shortcomings make it difficult to re-order the lanes between a pair of vectors.

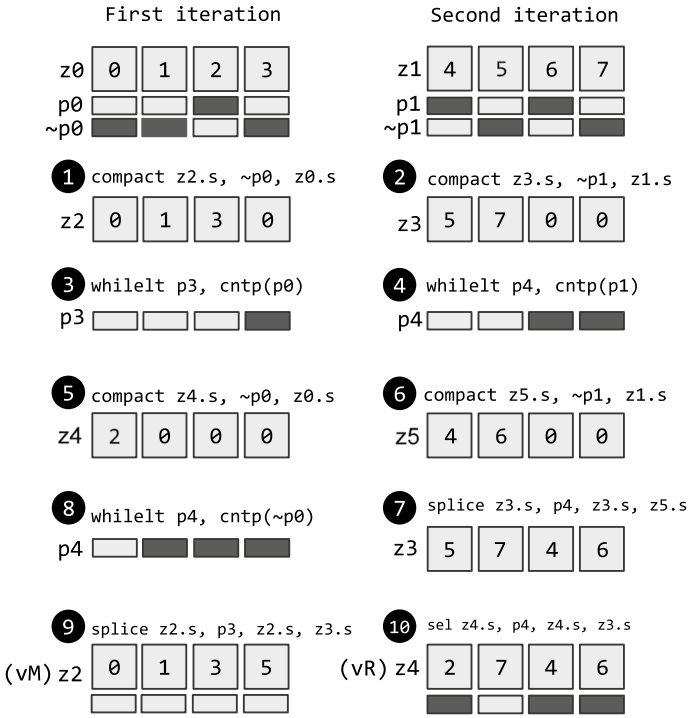


Fig. 8 Visual example of the ALC permutation implemented in SVE. Active lanes are shown as light colored squares while inactive as dark colored squares. For brevity, we use `cntp` as a function but requires another instruction that returns the count of active lanes. In addition, we use `~ p0` to represent the logical `NOT` of predicate register `p0`. In 1, once compacted, there are additional zero values appended at the end of the vector which are empty lanes rather than a valid zero index. These are not problematic when used as the predicate will prevent any instruction from using the zeros at the end

Listing 8: Proposed instruction to support the active-lane consolidation permutation in SVE

```
// Two-register group
consolidate {z0, z1}, {p0.t, p1.t}

// Three-register group
consolidate {z0, z1, z2}, {p0.t, p1.t, p2.t}

// Four-register group
consolidate {z0, z1, z2, z3}, {p0.t, p1.t, p2.t, p3.t}
```

Listing 9: Proposed extension of `tbl` to perform an multi-vector table lookup

```
1 // Two-register group
2 tbl z0.t, {z0, z1}, zN.t
3
4 // Three-register group
5 tbl z0.t, {z0, z1, z2}, zN.t
6
7 // Four-register group
8 tbl z0.t, {z0, z1, z2, z3}, zN.t
```

This section proposes two additional instructions to SVE that introduce the active-lane-consolidation permutation as an instruction and support for the multi-register-table permutation. Listing 8 shows the proposed syntax for this new instruction `consolidate` illustrating the consolidation of the adjacent register group { `zN, zN+1 ... zY`} based on the active lanes given by the predicate group {`pN.t, pN+1.t .. pY.t`} with an element size of *t*. Similar to the fixed-length `tbl`, the input operands for the proposed consolidation instruction are provided as a list of consecutive vector registers to allow consolidation of more than two registers. Following SVE's destructive operation scheme, the input register group is also used as output.

Alternatively, a design could extend the existing `compact` instruction to operate on a register group because the semantics of `compact` are very similar to the semantics of `consolidate`. Such extension would need to change the semantics of `compact` so that the inactive lanes are retained instead of being zeroed as the `compact` instruction currently does.

The second shortcoming of SVE in the context of ALC is the lack of an indexed permutation on a scalable vector. In the current SVE design, `tbl` takes a single vector register as input while in Advanced SIMD the fixed-length version of `tbl` allows up to four consecutive registers as input. Listing 9 shows a proposed extension of `tbl` where the last vector operand is a vector of indices into the register group defined within the curly braces. The proposed `consolidate` operation could also be used to achieve the same result but requires a predicate group to be supplied to the instruction.

5 Iterative ALC

A limitation of relying on the unrolling of vectorized loops to apply ALC is that consolidation will consider active lanes only within the vectors exposed by unrolling. Practically, there may be many more opportunities to form a uniform vector if not limited to consolidating consecutive iterations. This section presents Iterative ALC, a code transformation that overcomes this limitation by consolidating active lanes across an arbitrary number of iterations of a loop. The insight that led to the design of this Iterative ALC transformation is that the merged vector can persist

across loop iterations and consolidate active lanes until the merged vector becomes uniform, at which point it can be processed with 100% utilization.

Figure 9 shows an example of Iterative ALC applied to a loop with an `if-else` statement. The ALC permutation primitives and their applications are detailed in Sect. 3. In the figure, a thin line indicates a control line and a thick line indicates that all the lanes of the vector are used in the operation. Greyed lines and blocks indicate code that is not executed in this example. Iterative ALC initializes the merged vector using the first iteration of the vectorized loop ①. In each iteration, an *any* BOSCC checks for active lanes in the incoming iteration ② to determine if the ALC permutation should be applied ③. If the incoming vector is already uniform then ALC is not applied as a BOSCC can already exploit this vector. The ALC permutation produces the updated merged vector ④ that now contains any active lanes that were present in the incoming vector and also produces remainder vector ⑤. In iteration 1, after ALC, the remainder vector ⑤ is uniformly inactive so that the BOSCC check fails and the `else` block ⑥ is executed. In iteration 2, the *any* BOSCC ⑨ on the remainder vector ⑧ succeeds, revealing that not all active lanes could be moved into the merged vector ⑦ and thus the merged vector must now be uniform. The `then` block ⑩ can then be executed with 100% utilization. Once the uniform merged block is executed, the remainder vector in that iteration is set as the merged vector for the next iteration ⑪ as illustrated in iteration 3.

Iterative ALC is most beneficial in loops where a single condition needs to be evaluated to determine the flow of control—loops that contain a simple `if-else` control flow or with a single `if` statement rather than loops containing long `if-else-if` statements—because in the presence of more complex control flow iterative ALC would have to save all lanes of operands appearing in each control path. In addition, if ALC is performed to consolidate active lanes, all saved vector operands must also be permuted to correct the order, further adding to the complexity of operand merging. In the case of a single `if` statement with no `else` block, execution can be completely bypassed. In addition, because there is no need to retain inactive lanes, the ALC permutation shown in Listing 6 can be simplified. This simplification is not shown in detail but involves removing instructions in Listing 6 dealing with retaining the inactive lanes of the vectors being consolidated.

In the first transformation to enable ALC, described in Sect. 3.2, the number of active lanes available to be consolidated is limited by the unrolling factor. The unrolling transformation uses an unrolling factor of two because the combinatorial complexity of the ALC permutation increases with the unrolling factor. Larger unrolling factors are also limited by the number of vector registers available because unrolling increases register pressure. In contrast, iterative ALC is able to consolidate active lanes of vectors from any subsequent iteration of the vectorized loop, leading to better utilization at any vector length without additional register pressure.

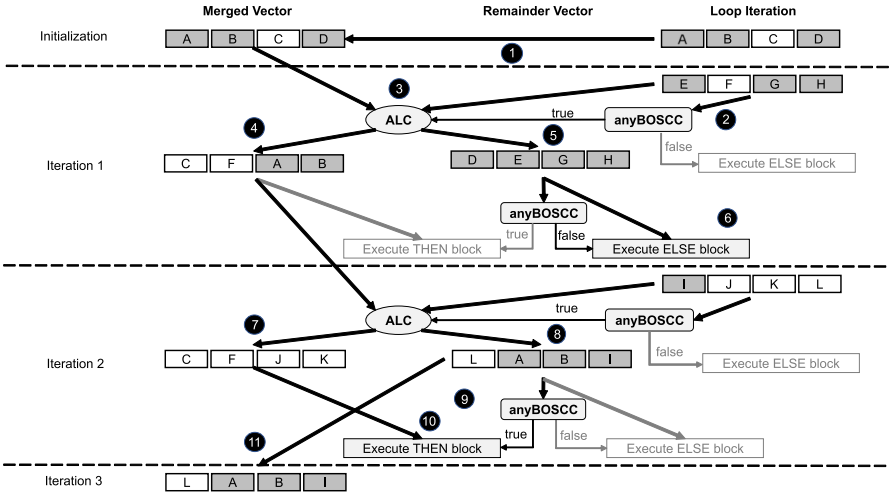


Fig. 9 Example of iterative ALC

6 Evaluation

This benchmark-based study assesses the opportunities to apply, and potential benefit of, ALC and iterative ALC in future programs. Using the SPEC CPU 2017 benchmark suite, this study tries to answer the following questions:

Q1: How many opportunities exist for applying ALC and iterative ALC in the SPEC CPU 2017 benchmark suite?
Q2: Can the ALC permutation, through loop transformations, be used to non-trivially increase vector utilization in loops that experience divergent control flow?
Q3: What is the overhead of performing the ALC permutation and index-based inter-register permutation?

6.1 Methodology

In order for a loop to contain an opportunity to apply ALC it must: 1. Not have any loop-carried dependencies; 2. Contain a block that is terminated by a conditional branch; 3. Not contain function calls. To answer **Q1** we designed a static analysis that detects loops with these properties. Table 1 shows the total number of loops found and the number of loops that meet the ALC conditions in each benchmark of the SPEC CPU 2017 benchmark suite compiled with vectorization, loop unrolling disabled and function inlining enabled. The same static analysis answers two additional questions of interest: how many conditional branches each loop contains; and how large are the executed blocks. Figure 10a shows a histogram of the number of conditional branches in ALC opportunity loops. Figure 10b shows a histogram of the number of LLVM IR operations in blocks inside the loop. The counts used to

construct the histograms in Fig. 10 exclude the latch, exit, and header blocks whose predecessors are terminated by a conditional branch. This analysis reveals that a large portion of the loops to which ALC can be applied contain few conditional branches and the conditional blocks contain a small number of operations.

For a profitable application of ALC, a loop also must exhibit divergent control flow during its execution. Therefore, the dynamic behavior of a loop also has to be examined. The remainder of this Section investigates four loops reported by the static analysis whose runtime branching behavior is divergent. These loops illustrate how ALC and iterative ALC affect the vector execution and performance. Divergency was confirmed by running the benchmark and inspecting the branch directions at runtime. The criteria for divergency were simply a non-uniform list of branch outcomes. When implemented inside a production compiler, the application of ALC must also check for other factors, such as non-vectorizable statements, that may prevent the application of these transformations.

As of writing, the only SVE-enabled processor available is the Fujitsu A64FX [23] that we are unable to access. Thus, this performance estimation uses the Arm Instruction Emulator (ARMIE) [24] running on the non-SVE-enabled AArch64 hardware. ARMIE uses DynamoRIO [25] to perform dynamic binary translation. ARMIE replaces SVE instructions with an equivalent sequence of scalar AArch64 instructions. Because the SVE instructions are being emulated, wall-clock runtime cannot be used as a result.

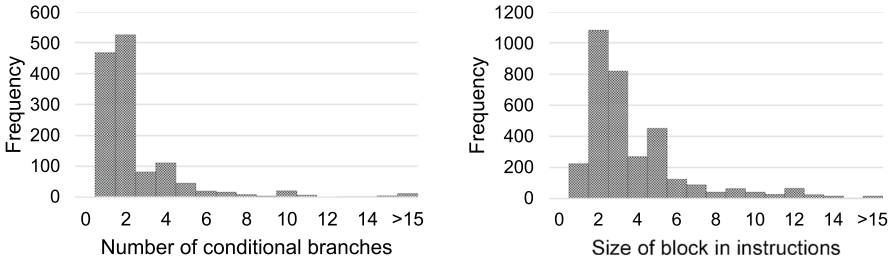
The goal of ALC is to increase uniformity in vectors to improve the efficacy of traditional compiler optimizations such as BOSCCs. BOSCCs elide the execution of redundant instructions by exploiting uniform vectors and thus reducing the dynamic instruction count. ARMIE collects these key measurements and enables an assessment of the efficacy of ALC. Dynamic runtime statistics collected through ARMIE include dynamic instruction count, control-flow-path frequency, and vector-instruction utilization. These statistics are used to compare ALC to the previous state of the art. Each version of the loop is compiled using Clang 10 with the `-fno-vectorize -fno-slp-vectorize -fno-unroll-loops` compiler flags. Emulated executions take much longer to complete than native executions, therefore, the `TRAIN` workload of the SPEC CPU 2017 benchmark suite is used for all ARMIE-based measurements. A comparison of the frequency of execution of the branch instructions and of the branch outcomes with executions of the `REFRATE` workload revealed that they are very similar to the `TRAIN` workload.

6.2 Case studies

The four test cases used in this study are loops taken from the following C/C++ SPEC CPU2017 benchmarks: NAB, LBM, and MCF. Open-source production compilers' implementations of SVE are in very early development. Until the requisite infrastructure necessary to write an automatic compiler pass—which requires effort from a team of developers—is available, the potential for applying ALC can

Table 1 Loops where ALC may be applicable discovered by the static analysis described in Sect. 6.1

	Total	Applicable
500.perlbench	4667	11
502.gcc	17,658	115
505.mcf	85	2
520.omnetpp	2170	1
523.xalan	10,886	24
525.x264	1127	5
531.deepsjeng	148	8
541.leela	189	8
557.xz	247	0
508.namd	1449	0
510.parest	5486	35
511.povray	2450	35
519.lbm	2450	28
526.blender	19	4
538.imagick	36,337	928
544.nab	2034	142
Sum	85,230	1322



(a) Conditional branches in each loop.

(b) Number of instructions in each block of the loop.

Fig. 10 Results of a static analysis on the loops matching the basic criteria for ALC to be applicable in the SPEC CPU 2017 benchmark suite

be assessed by rewriting each loop by hand using the ARM C language extensions (ACLE) [22]. This laborious process limits the number of case studies reported.

LBM LBM is a fluid-dynamic benchmark that implements the Lattice-Boltzmann method to simulate incompressible fluids in 3D [26]. The case study from LBM is extracted from the function `StreamCollideTRT`, which takes 97% of the runtime when LBM is run on the `TRAIN` workload. Figure 11 shows the two main paths of control in this loop, B_1 performs 19 loads and stores while B_2 performs 49 loads and 19 stores. The block B_2 also performs many multiply, add and subtract arithmetic operations. The step increment of the loop induction variable is not unitary. Therefore, after vectorization, load and store operations in B_1 and B_2

are translated into gather loads and scatter stores. The predicate computation for the condition C_1 in the vectorized version is simple because it depends only on a load. Also of importance, the statements inside the conditional blocks do not depend on values computed outside and therefore the ALC loop transformation does not need to permute operands.

NAB The NAB (Nucleic acid builder) is a molecular-dynamic program in the SPEC CPU2017 benchmark suite [26]. The case study from NAB is from the `egb` function and contains parallel hot loops with complex control flow. The control-flow excerpt for the first of the three loops is shown in Fig. 11a and was also used in the study that evaluated the effect of simple BOSCCs on this loop [17].

MCF_1 The first case study from the MCF benchmark, `MCF_1`, is from the `flow_cost` function. The MCF benchmark solves a network-flow problem to compute schedules for public transportation [26]. The loop in this case study takes around 1% of the execution time of the benchmark.

MCF_2 The second case study from the MCF benchmark, `MCF_2`, is found in the `read_min` function. This loop calls the function `getArcPosition`, which contains an `if-else` statement and is inlined into the loop during compilation.

6.3 Loop transformations

For these case studies, each loop is vectorized using ACLE intrinsics to insert BOSCC instructions and apply the unrolling ALC and iterative ALC transformations. The different versions of the loop implemented and evaluated are as follows:

ifcvt: In this baseline the loop is vectorized with IF-conversion and no BOSCC instructions are inserted. This version is similar to how the LLVM and GCC vectorizers currently transform these loops.

boscc: After vectorization and IF-conversion, additional BOSCC instructions are inserted, where appropriate, to optimize execution.

alc_unroll: The unrolling ALC transformation described in Sect. 3.2 is applied to the vectorized and IF-converted loop to consolidate active lanes of a conditional block.

alc_iter: In applicable benchmarks, the iterative ALC transformation described in Sect. 5 is applied to the loop. Iterative ALC was only applied to the LBM and MCF_1 loops because the long `if-else-if` statement in the NAB kernel and the small conditional blocks in MCF_2 prevented application on these kernels.

For the comparison with **boscc** one must establish the location in the execution path where BOSCC instructions should be placed and the type of instruction—`all`, `any` or `none`—to be placed. Both factors can greatly affect performance. At the time, there is no automated compiler analysis to determine a suitable placement of BOSCC instructions into a vectorized and if-converted control flow graph. Thus, in order to present a fair comparison with the ALC approaches for this study, we empirically vary the placement and type of BOSCC and use the best-performing kernel as a baseline for the comparison. For instance, the NAB loop contains an `if-else-if` statement that creates five conditional blocks in the IR. As shown

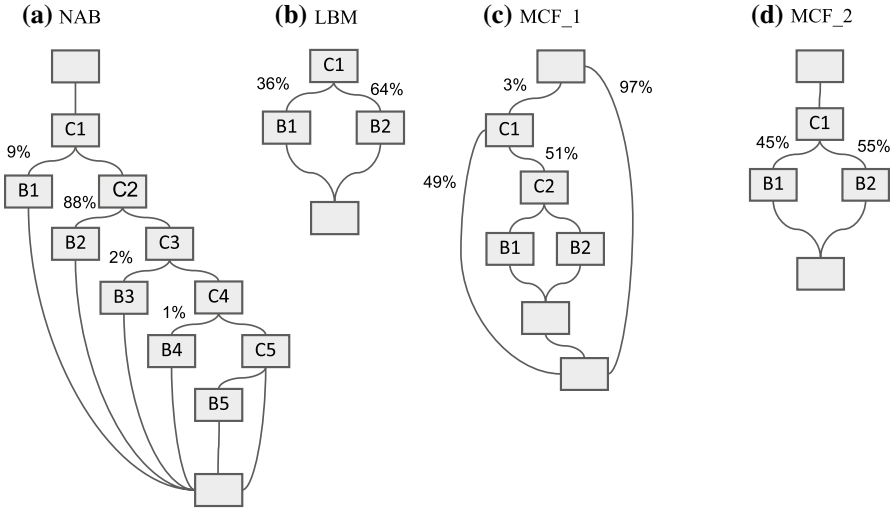


Fig. 11 Control flow graphs of the body of the loops of the benchmarks. Percentages attached to edges indicate the runtime probability the branch is taken in the TRAIN workload

in Fig. 11a, block B_2 is by far the most frequently executed conditional block in the NAB kernel. Thus, the insertion of an *all* BOSCC on the condition computed in C_2 that predicates B_2 leads to the execution of B_2 only if all lanes are active and results in good performance. However, another performant solution is to insert an *any* BOSCC *after* B_2, before the execution of B_3 to elide execution of the remaining linearized blocks in the case where all lanes were active for either B_2 or B_1. In this study, we run experiments for both and report the best-performing kernel. The locations in the control flow graphs where BOSCC instructions are inserted for each case study are shown as dashed edges in Fig. 12d.

For **alc_unroll**, future compiler analysis will be needed to decide which conditional block should be consolidated. In most cases, consolidating on the most frequently executed block is a good heuristic. However, there are cases where consolidating a less frequently taken block, such as B_1 in the LBM benchmark, is more beneficial. In this study, a similar empirical approach to the one used for **boscc** decides which block to consolidate for **alc_unroll**.

6.4 Results

Figure 12 reports the dynamic instruction-count reduction in the region-of-interest (ROI) for each benchmark for the three code transformations. The baseline is *ifcvt*. In comparison to the *boscc* kernels, the instruction reduction in *alc_unroll* is greater for longer vectors in the LBM and NAB case studies (12% and 39% at 2048 bits). For the MCF_2 case study, no results are reported because all three code transformations failed to reduce dynamic instruction count over baseline.

In the **alc_unroll** kernels, the control flow path which consolidates active lanes of vectors is followed only when there are enough active lanes to create a uniform vector and none of the input vectors are already uniform before the ALC permutation. To handle situations where uniform vectors are encountered before ALC, BOSCCs are inserted in the fallback path that executes the regular if-converted code. BOSCCs inserted in the fallback path can account for a large portion of the instruction reduction in the unrolling ALC versions. In fact, at lower vector lengths, such as 128-bits and 256-bits, most of the instruction reduction reported in Fig. 12 is due to these BOSCCs.

A downward trend emerges in the graphs as VL increases because a larger portion of the vectors in the loop become divergent causing BOSCCs to lose effectiveness. In the **alc_unroll** kernels, the effect is lessened because the divergent vectors are consolidated into a uniform vector, leading ALC to out-perform the **boscc** kernels in the LBM and NAB benchmarks after a certain vector length (1024 bits in LBM and 512 bits in NAB).

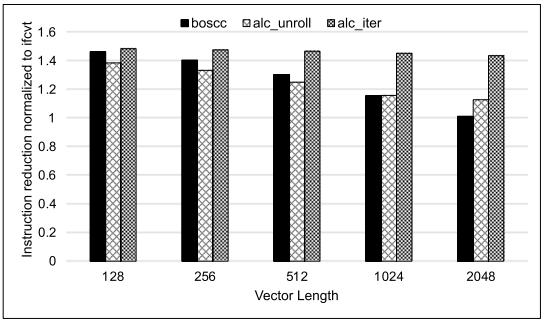
There is a clear correlation between lower dynamic instruction count and the loop kernels that execute more efficiently, i.e., higher vector utilization reported in Fig. 13. In the LBM loop, the unrolling ALC transformation only begins to yield instruction reduction (Fig. 12a) over the *boscc* kernel once there is a clear gap between utilization, as seen in vectors greater than 1024 bits on **alc_unroll**, compared with the **boscc** kernel in Fig. 13a. This trend is also observed in the NAB loop between **boscc** and **alc_unroll**.

alc_unroll finds many opportunities to consolidate vector lanes in MCF_1 (21% of iterations consolidate vectors at 2048 bits). However, it fails to out-perform **boscc** regardless of vector length because bypassing the vectorized code for the small basic blocks does not amortize the overhead introduced by performing the ALC permutation.

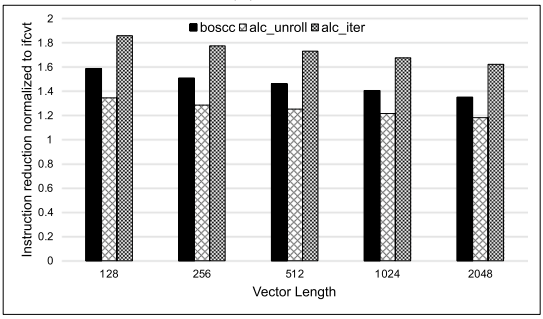
6.5 Iterative ALC

When **alc_unroll** encounters a vector that is uniform prior to the ALC permutation, the fallback path is taken because consolidation does not benefit this case. This fallback path contains IF-converted code with BOSCCs applied to exploit the existing uniform vector as done for **boscc**. When one of the two vectors exposed by unrolling is uniform and the other is divergent, the divergent vector must be processed by the inefficient if-converted code similar as is done for **ifcvt**. When both candidate vectors are divergent but do not contain enough active lanes to consolidate into a uniform vector, both vectors are processed by the inefficient if-converted code in the fallback path. **alc_iter** addresses these inefficiencies by consolidating active lanes from several iterations until a uniform vector is formed instead of being limited to consolidating lanes only from the vectors exposed by unrolling.

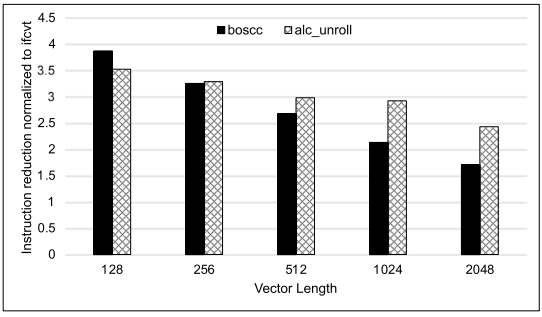
Iterative ALC out-performs both **boscc** and **alc_unroll** kernels in the two loops where it was applied: LBM and MCF_1. We only apply iterative ALC to these two loops as the simple control flow structures present in the kernels make the implementation of iterative ALC straightforward. Iterative ALC was not applied on the



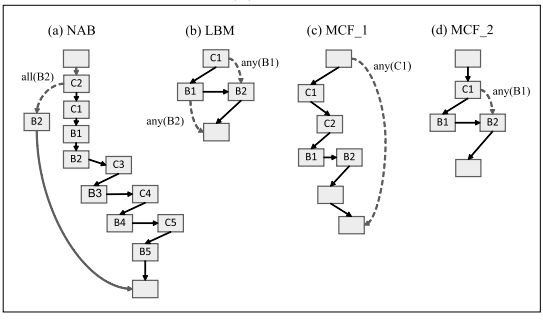
(a) LBM



(b) MCF_1



(c) NAB



(d) If-converted CFGs with BOSCCs

Fig. 12 Reduction in dynamic instruction count of loops optimized with BOSSCs (boscc) or the ALC transformations (alc_unroll, alc_iter) over the kernel with only vectorization and `if`-conversion (ifcvt) applied. Figure 12d shows the control-flow graphs of the kernels with the locations of the BOSSC branches that are indicated by a dashed line

NAB kernel as the nature of iterative ALC requires that all operands used within every conditional block be saved for later permutation. The five-long `if-else-if` cascade present in the NAB loop leads to a prohibitive state-saving space requirement. Furthermore, iterative ALC is less applicable to loops with long `if-else-if` cascades because an automatic compiler pass to perform the transformation would require a well-informed decision about which conditional block to consolidate compared to loops with simple single-`if` constructs.

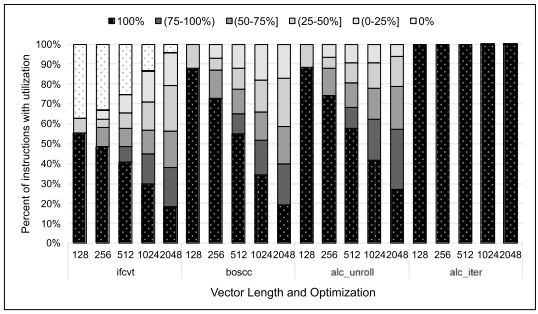
Overall, fewer instructions are executed by **alc_iter** because vectors are utilized more efficiently. Where the **boscc** and **alc_unroll** would sub-optimally process a vector with a partially active predicate, **alc_iter** trades inactive lanes in this vector with active lanes encountered in other iterations of the loop.

The MCF_1 case study contains a nested if-statement that sparsely takes the branch to block C_1 shown in Fig. 11. BOSCC instructions are not effective for longer vectors in this case because the long vectors will often contain a few active lanes and cause the any-true BOSCC to succeed so that the branch to C_1 is taken. This gradual degradation of **boscc** due to it executing instructions with "near-empty" predicates as vector length increases is observed in Fig. 13b. The sparsity of C_1—taken 5% of the time—leads alc_unroll to struggle to find opportunities to merge vectors from consecutive iterations. In comparison, iterative ALC steals active lanes from several iterations to create a uniform merged vector with active lanes. The consolidation of active lanes into the merge vector leaves a uniform remainder vector with only false lanes whose execution can be completely bypassed because the control-flow structure of the kernel only contains a single `if` statement with no `else`. The result is that **alc_iter** outperforms both **boscc** and **alc_unroll** in both vector utilization (Fig. 13b) and dynamic instruction reduction (Fig. 12b).

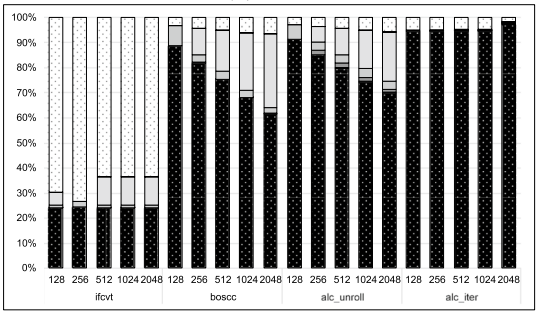
6.6 Under-utilization of vector instructions

Figure 13 shows the percentage of vector instructions that are executed with a certain predicate utilization. Darker shades indicate more lanes of the predicate are active while lighter shades indicate less. The **ifcvt** kernel shows the results from the version of the loop vectorized with naive IF-conversion and is similar to how LLVM and GCC would currently vectorize these loops. Figure 13 shows that a very large portion of vector instructions execute with a predicate that is either all-false or has low utilization. This illustrates the poor execution efficacy that traditional IF-conversion leads to especially when control divergence becomes more prevalent as the vector length increases.

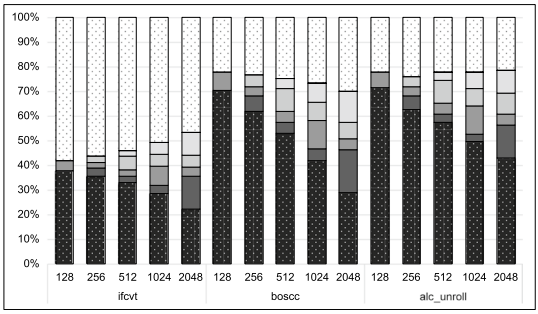
The results in Fig. 13 indicate that when properly placed, BOSCC instructions can effectively avoid executing instructions with all-false predicates. For instance, an `all` BOSCC inserted on B_2 in NAB and an `any` BOSCC inserted on B_1 in LBM result in the largest instruction reduction because B_2 is the hottest block in NAB and B_1 is the least frequent block in LBM. However, a disadvantage of both **boscc** and **alc_unroll** is that they both depend on a yet-to-be-created compiler analysis. Such analysis has to determine the place and type of BOSCC instructions to insert for **boscc** and which block to consolidate for **alc_unroll**. These decisions depend on



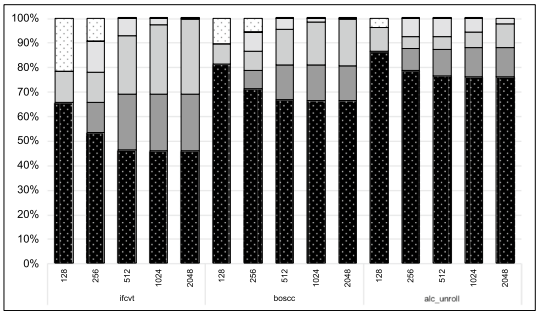
(a) LBM



(b) MCF_1



(c) NAB



(d) MCF_2

Fig. 13 Percent divergence of vector instructions. Dotted black bars indicate 100% utilization while dotted white bars indicate 0% utilization. Shaded bars indicate partial utilization with darker shades representing higher utilization

accurately predicting the branch probabilities at runtime. For some programs, such probabilities depend on the workload and thus vary from run to run.

In comparison, **alc_iter** fills the merged vector as it encounters active lanes in the loop and therefore it does not depend on accurate branch-outcome information. For instance, the block being consolidated in Fig. 13a is the lesser executed block B_1. Even so, **alc_iter** achieves perfect utilization and significant instruction reduction over the best results for **boscc** and **alc_unroll**.

6.7 Overhead of the ALC permutation

This section addresses question **Q3**. As discussed in Sect. 4 the ALC permutation requires at most fifteen instructions while the inter-register permutation requires three instructions per operand that needs permuting. To be profitable, the ALC loop transformations must amortize the cost of executing instructions to perform these permutations.

Table 2 shows the instruction count for the primary control flow paths in the loop kernels. As illustrated in Fig. 12d BOSCC branches introduce control flow for **boscc**, **alc_unroll** and **alc_iter** to allow bypassing sections when all the lanes of a vector are inactive. If any of the lanes are active, the BOSCC branch is not taken and the entire sequence of partially active predicated vector instructions must be executed. The number of instructions for each of these paths is reported under the columns *boscc not-taken* and *boscc taken* in Table 2.

The slight increase in the number of instructions from *ifcvt* to *boscc not taken* is indicative of the low overhead introduced by the BOSCC instruction. The drastic instruction-count reduction from *ifcvt* to *boscc taken* for the LBM, MCF_1 and NAB kernels underscores the inefficiency of applying only IF-conversion to the loop. This inefficiency results from *ifcvt* executing predicated code to account for all paths, including some which may be predicated by an all-false predicate.

In the **alc_unroll** kernels, there is a taken and a not-taken path. In the taken path, ALC is performed and thus, it includes instructions to perform the ALC and index-based inter-register permutations. In the not-taken path, ALC is not performed either because candidate vectors for consolidation are already uniform or because there is an insufficient number of active lanes to form a uniform vector. Note that **alc_unroll** executes two iterations of the loop while *ifcvt* to *boscc* execute a single one—this difference accounts for static instruction counts for **alc_unroll** being close to twice the amount reported in the **ifcvt** column.

For **alc_unroll** in MCF_2 the taken path is longer than the not-taken path and thus consolidation will result in performance degradation because several operands must be permuted. The issue is compounded by the small size of the basic blocks, which lowers the benefit of a BOSCC branch. Larger blocks would result in more savings from bypassing execution and would amortize the operand permutation costs. For kernels exhibiting similar characteristics where many operands need to be permuted to correctly execute a small consolidated block, it is not beneficial to consolidate execution of vectors through the unrolling ALC transformations.

Table 2 Static instruction counts for the control flow paths in each kernel

Kernel	**ifcvt**	**boscc**		**alc_unroll**		**alc_inter**	
		Not-taken	Taken	Not-taken	Taken	Filled	Continue
lbm	583	589	74	1212	114	96	546
nab	204	205	98	470	331		
mcf_1	25	32	15	74	66	48	21
mcf_2	45	46	40	90	130		

With the proposed ISA design changes described in Sect. 4.4 both the ALC permutation and the index-based inter-register permutation would require a single instruction each. Such change would significantly reduce the overhead of ALC. For instance, for MCF_2, the overhead could be reduced to seven instructions: one for ALC and six for index-based inter-register permutation of operands for the consolidated block and for the remainder block. Significant overhead reduction, such as this one for MCF_2, will enable the use of ALC for loops with small block sizes. This is especially interesting given the findings presented in Fig. 10 that show that small conditional blocks, containing only two to eight instructions are most common.

The **alc_iter** kernels contain two paths, *filled* and *continue*. The *filled* path processes the merged vector once it becomes full, this is illustrated in iteration 2 of Fig. 9. The *continue* path represents the path that control follows when the ALC permutation is performed but the merged vector does not become full. In the case of a single `if` statement with no `else` block, such as in the MCF_1 benchmark, no additional code is executed. Furthermore, because there is no need to retain the inactive lanes, a simplified version of the ALC permutation, that requires fewer instructions, can be used. These characteristics make iterative ALC very appealing to optimize loops with lone `if` statements. In contrast, the LBM benchmark contains an `if-else` statement so that the *continue* path executes code to process the lanes that were inactive for the `if` block. In, MCF_1, 27 of the 48 instructions present in the *filled* path are related to the actual execution of the conditional block. The remaining 21 instructions execute the ALC permutation to consolidate active lanes. The difference between the 15 required instructions presented in Sect. 5 and the 21 listed here is due to imperfections in the compiler's code generation. Even with the large number of instructions required to perform the ALC permutation, the iterative ALC kernel outperforms the best *boscc* kernel because of the large increase in vector utilization provided by iterative ALC.

7 Related work

Control flow has long been an obstacle for SIMD execution. IF-conversion is the canonical method that enables vectorization of codes containing control flow and works by linearizing the control flow graph so that all possible paths of control are executed [2]. IF-conversion linearizes execution by associating a predicate to each

block. Blocks that would execute in the original code receive a true predicate and produce results while the remaining blocks do not produce results. In loops suffering from control flow divergence, IF-converted and vectorized code leads to underutilized vector units and inefficient execution; in the worst case, a vector instruction will execute with all lanes inactive, producing no result while still occupying processor resources. Shin et. al. use Branch-on-superword-condition-codes (BOSCCs) to alleviate vector under-utilization by inserting BOSCC instructions that elide execution of unnecessary instructions [4, 5].

A technique to prevent the deterioration of SIMD utilization consists of partially linearizing the control flow [17]. This algorithm retains branches proven to be uniform by divergence analysis [27]. Partial linearization also inserts uniform branches with the same semantics as BOSCCs, called BOSCC gadgets, which are later lowered to their respective BOSCC instruction in the vectorized code. Moll et. al describe the insertion of `any` BOSCC gadgets to bypass execution of infrequently executed blocks and evaluate its effect with Intel AVX-512 and Arm Advanced SIMD processors in the NAB benchmark yielding a runtime improvement of up to 30%.

Warp-coherent-condition vectorization (WCCV) extends partial control-flow linearization by proposing methods to detect warp-coherent conditions and to transform loops containing them [21] . Warp-coherent conditions are conditions that, when vectorized, exhibit similar behavior between lanes. In contrast to the use of `any` BOSCCs in partial linearization, WCCV inserts `all` BOSCCs in a code transformation that executes only the block for which the condition is true, thus bypassing redundant code when the vector is uniform or "warp-coherent."

Both partial control-flow linearization and WCCV suffer when faced with heavily divergent control flow because BOSCCs fail to optimize the resulting divergent vectors. This paper presents the ALC vector permutation that consolidates active lanes between two vectors. The permutation creates more cases of uniform vectors that `all` BOSCCs can detect and optimize during runtime. ALC is integrated into two code transformations to expose additional vectors to consolidate: unrolling and iterative. The unrolling ALC transformation unrolls the loops by a factor of two to consolidate the lanes from different iterations of the vectorized loop. Lanes that are not consolidated into the uniform vector, called the remainder vector, need to be handled and can be processed through the IF-converted graph. Also, `any` BOSCCs can be inserted within the IF-converted graph to further optimize execution. With iterative ALC, active lanes from multiple iterations can be consolidated into a single vector for uniform execution.

The concept of increasing vector utilization through “stealing” lanes already exists and has been proposed for other use cases and architectures. Lang et al [28] present a method to refill vector lanes in vectorized database queries with the `permute`, `compress` and `expand` instructions in Intel AVX-512. This work did not address the issue when inactive lanes must be saved for their own processing. In contrast, the ALC permutation presented in this paper swaps inactive lanes in one vector for active lanes in another vector thereby saving the inactive lanes if they are still required. In addition, Lang et al.’s work focused on applying their methods to database queries and implemented them in AVX-512; the ALC transformations

presented in this work are applicable to any vectorizable loop with divergent control flow and we present an implementation using ARM's SVE.

Both Intel AVX-512 and ARM SVE are CPU extensions and hence, the ALC permutation and loop transformation target SIMD execution on CPUs only. However, the concept of reorganizing lanes to increase vector utilization is analogous to the re-organization of threads in a warp to increase SIMD utilization in GPUs. Hardware-based approaches that perform fine-grain lane reorganization on GPUs [29, 30] dynamically create new warps to mitigate control-flow divergence. Software approaches [31] have limited applicability as they must emulating moving lanes by copying data to shared memory. In general, reorganizing lanes on GPUs is problematic due to the large amount of state associated with each lane in the SIMT paradigm.

Barredo et al. present the Compaction/Restoration (CR) technique to improve SIMD utilization [3]. CR proposes a hardware mechanism on top of the existing vector processing unit (VPU) to catch "compactable" instructions as they move through the out-of-order pipeline. Multiple similar instructions are then compacted into a single "dense" instruction that is issued to the VPU for execution. Such a scheme has the potential to improve vector efficiency in many loops without prior analysis and intervention by the compiler. However, because CR is implemented in hardware, the distance in which active lanes can be compacted is limited and thus it is unable to operate in a similar manner to the iterative ALC transformation introduced in this work where lanes can be consolidated from many loop iterations.

The ARM Scalable vector extension was introduced to address issues with past vector extensions such as ISA disorganization and scalability [11] and as a solution to drive performance in numerous application domains in the face of demanding power requirements [32, 33]. The popularity of SVE is increasing and mainstream compilers such as GCC and LLVM are actively working to implement support [34]. Research thus far has focused on using SVE to accelerate applications in specific workloads such as stencil codes [35] and image processing pipelines [36] and evaluates the resulting performance. ARM SVE is not the only ISA to re-adopt the VLA architecture. RISC-V [13] is also introducing scalable vectors into their ISA for which production hardware has already been produced [37].

8 Conclusion

Vectorizing loops with control flow is a long-standing problem. IF-conversion offers a solution to vectorize these loops but leads to inefficient execution in loops with divergent control flow. The trend of increasing vector length exacerbates the inefficiency caused by control-flow divergence. This paper presents a novel vector permutation, Active-Lane Consolidation, to consolidate active lanes between two divergent vectors and facilitate the formation of a uniform vector. Traditional BOSCC branches can then exploit this opportunity by bypassing the execution of unnecessary predicated vector code, leading to an increase in vector utilization. The paper

presents two loop transformations to illustrate the use of ALC: unrolling ALC and iterative ALC.

Performance prediction based on case studies of the ALC transformation on a set of four kernels found in the SPEC CPU 2017 benchmark suite indicates that ALC has significant potential to increase vector utilization and to decrease dynamic instruction count.

Acknowledgements This research was funded by the University of Alberta Huawei Joint Innovation Collaboration (UAHJIC) and by the National Sciences and Engineering Research Council (NSERC) of Canada. We thank Giancarlo Pernudi Segura for his great assistance creating some of the assembly-level coding for the case studies.

References

1. Monroe D (2020) Fugaku takes the lead. Commun ACM 64(1):16–18
2. Allen, JR, Kennedy, K, Porterfield, C, Warren, J (1983) Conversion of control dependence to data dependence. In: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on principles of programming languages, pp 177–189
3. Barredo A, Cebrian JM, Moretó M, Casas M, Valero M (2020) Improving predication efficiency through compaction/restoration of simd instructions. In: 2020 IEEE international symposium on high performance computer architecture (HPCA), pp 717–728
4. Jaewook S (2007) Introducing control flow into vectorized code. In: 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), pp 280–291. IEEE
5. Shin J, Hall MW, Chame J (2009) Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. Microprocess Microsyst 33(4):235–243
6. Flynn MJ (1972) Some computer organizations and their effectiveness. IEEE Trans Comput C–21(9):948–960
7. Intel Corporation (2021) Intel AVX-512. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html
8. ARM Corporation (2021) ARM Advanced SIMD. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon
9. Arm Limited (2021) Arm®Architecture Reference Manual Armv8, for Armv8-A Architecture Profile
10. Russell RM (1978) The CRAY-1 computer system. Commun ACM 21(1):63–72
11. David Patterson (2017) SIMD Instructions Considered Harmful. https://www.sigarch.org/simd-instructions-considered-harmful
12. Arm Limited (2021) Arm®Architecture Reference Manual Supplement The Scalable Vector Extension (SVE), for Armv8-A
13. RISC-V® International Members (2021) The RISC-V "V" vector extension. version 0.10 (Visited on April 26, 2021). https://github.com/riscv/riscv-v-spec/releases/download/v0.10/riscv-v-spec-0.10.pdf
14. Sreraman N, Govindarajan R (2000) A vectorizing compiler for multimedia extensions. Int J Parallel Prog 28(4):363–400
15. Kennedy K, Allen JR (2001) Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., Massachusetts
16. Wolfe MJ (1995) High performance compilers for parallel computing. Addison-Wesley Longman Publishing Co. Inc, New York
17. Moll S, Hack S (2018) Partial control-flow linearization. ACM SIGPLAN Notices 53(4):543–556
18. Allen F, Cocke J (1971) A catalogue of optimizing transformations. Prentice-Hall, New Jersey
19. Anantpur J, Govindarajan R (2014) Taming control divergence in gpus through control flow linearization. In: Albert C (ed) Compiler construction. Springer, Berlin Heidelberg, pp 133–153

20. Sun H, Gorlatch S, Zhao R (2018) Refactoring loops with nested ifs for simd extensions without masked instructions. In: European Conference on Parallel Processing, pp 769–781. Springer
21. Sun, H, Fey F, Zhao J, Gorlatch S (2019) WCCV: improving the vectorization of IF-statements with warp-coherent conditions. In: Proceedings of the ACM International Conference on Supercomputing, pp 319–329
22. ARM (2020) The arm C language extensions https://developer.arm.com/architectures/system-architectures/software-standards/acle
23. Fujitsu Limited (2021) A64FX®Microarchitecture Manual. Version 1.4
24. ARM (2020) The ARM instruction emulator. https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator
25. Bruening D, Amarasinghe S (2004) Efficient, transparent, and comprehensive runtime code manipulation. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering
26. SPEC (2021) SPEC2017 Benchmark overview. https://www.spec.org/cpu2017/Docs/overview.html
27. Coutinho B, Sampaio D, Pereira FMQ, Meira Jr W (2011) Divergence analysis and optimizations. In: 2011 International Conference on Parallel Architectures and Compilation Techniques, pp 320–329. IEEE
28. Lang H, Passing L, Kipf A, Boncz P, Neumann T, Kemper A (2020) Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. VLDB J 29(2):757–774
29. Fung WWL, Sham I, Yuan G, Aamodt TM (2007) Dynamic warp formation and scheduling for efficient gpu control flow. In: 40th annual IEEE/ACM international symposium on microarchitecture (MICRO 2007), pp 407–420. IEEE
30. Fung WWL, Aamodt TM (2011) Thread block compaction for efficient simt control flow. In: 2011 IEEE 17th international symposium on high performance computer architecture, pp 25–36. IEEE,
31. Khorasani F, Gupta R, Bhuyan LN (2015) Efficient warp execution in presence of divergence with collaborative context collection. In: Proceedings of the 48th international symposium on microarchitecture, MICRO-48, pp 204-215
32. Stephens N, Biles S, Boettcher M, Eapen J, Eyole M, Gabrielli G, Horsnell M, Magklis G, Martinez A, Premillieu N et al (2017) The ARM scalable vector extension. IEEE Micro 37(2):26–39
33. Sato M, Ishikawa Y, Tomita H, Kodama Y, Odajima T, Tsuji M, Yashiro H, Aoki M, Shida N, Miyoshi I, et al (2020) Co-design for A64FX manycore processor and "Fugaku". In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–15. IEEE
34. Lovett (2021) SVE in LLVM. https://hps.vi4io.org/_media/events/2020/llvm-cth20_lovett.pdf
35. Armejach A, Caminal H, Cebrian JM, Langarita R, González-Alberquilla R, Adeniyi-Jones C, Valero M, Casas M, Moretó M (2020) Using Arm® scalable vector extension on stencil codes. J Supercomput 76(3):2039–2062
36. Cococcioni M, Rossi F, Ruffaldi E, Saponara S (2020) Fast deep neural networks for image processing using posits and arm scalable vector extension. J Real-Time Image Process 17:759–771
37. Chen C, Xiang X, Liu C, Shang Y, Guo R, Liu D, Lu Y, Hao Z, Luo J, Chen Z, et al (2020) Xuantie-910: a commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: industrial product. In: 2020 ACM/IEEE 47th annual international symposium on computer architecture (ISCA), pp 52–64. IEEE