PostSLP: Cross-Region Vectorization of Fully or Partially Vectorized Code

Vasileios Porpodas and Pushkar Ratnalikar
Intel Corporation, USA
{vasileios.porpodas, pushkar.v.ratnalikar}@intel.com

Abstract. Modern optimizing compilers rely on auto-vectorization algorithms for generating high-performance code. Both loop and straight-line code vectorization algorithms generate SIMD vector instructions out of scalar code, with no intervention from the programmer.
In this work, we show that the existing auto-vectorization algorithms operate on restricted code regions and therefore are missing out vectorization opportunities by either generating narrower vectors than those possible for the target architecture or are completely failing and leaving some of the code in scalar form. We show the need for a specialized post-processing re-vectorization pass, called PostSLP, that has the ability to span across multiple regions, and to generate more effective vector code. PostSLP is designed to convert already vectorized, or partially vectorized code into wider forms that perform better on the target architecture. We implemented PostSLP in LLVM and our evaluation shows significant performance improvements in SPEC CPU2006.

1 Introduction

Software applications increasingly rely on SIMD vector-hardware for high performance. This reliance on vector-units for performance has also led to the throughput improvements and refinements of vector instruction sets (ISAs), such as the Intel®AVX-512. Software developers have a few options for exploiting the full potential of vector-hardware: They can use target-specific intrinsics, they can call high-performance libraries, or they can use programming models like OpenMP [16].All of these approaches require additional effort by the software developer, and they can lead to non-portable code or code with non-portable performance. Compiler auto-vectorization aims at automatically converting scalar code into vector code, tuned for the target hardware.

There are two primary approaches for vectorizing scalar code. *Loop vectorization* widens the operations within a loop body [6,1,15], while *SLP vectorization* [8,25] replaces groups of isomorphic instructions with their corresponding vector instructions. Most production compilers including GCC [4] and LLVM [9] implement both.

The main motivation for this work is the observation that current Loop and SLP vectorization algorithms miss out vectorization opportunities and, in several cases, generate narrower vectors than are possible. Loop vectorizers, like the one in LLVM, are restricted to vectorizing within a set of consecutive iterations, combining consecutive iterations of a loop to a vector form. It does not check whether these could be combined with instructions from subsequent iterations that belong to the next region. Opportunities like these show up when the loop-body contains instructions of different bit-widths, as explained in Section 3.3.

The SLP vectorizer also operates on regions which terminate on one end at the *seed* instructions (stores to consecutive memory locations, reduction tree, etc.) and at the other end at either load or gathering points where isomorphism no longer holds. Current approaches do not consider cross-region vectorization, resulting in generation of smaller vectors, often leaving some code in scalar form.

In this paper we introduce a new post-vectorization technique called PostSLP that can successfully vectorize code that is not vectorized by existing state-of-the-art auto-vectorization algorithms. It specifically focuses on vectorization of operations across vectorized regions or at the boundaries of those regions. We show that that PostSLP can successfully form wider vectors out of either partially vectorized code (i.e., code with some scalars and some vectors), or out of fully vectorized code but with narrower vectors. This results in generation of wider vectors out of either partially vectorized code or already vectorized code but with narrower vectors. Our contributions include:

1. Highlighting a major weakness in existing SLP and loop-vectorization approaches with respect to their ability to maximize the vector length.
2. Proposing a new compiler post-vectorization pass, to be placed after both auto-vectorization passes in the pipeline, that vectorizes code (i) cross vector regions horizontally, and (ii) can seamlessly handle both vector and scalar instructions.
3. Evaluating our algorithm in an industrial compiler and showing that the proposed pass pipeline, with PostSLP, can consideralby improve performance of real-world workloads.

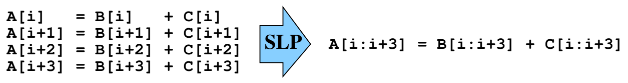
2 Background on Auto-Vectorization

Auto-Vectorization is a performance-critical optimization in modern compilers. Its goal is to replace scalar code with equivalent vector code, which has higher performance when the target architecture supports SIMD vector units. Modern compilers typically have two approaches to auto-vectorization,

1. Loop-based auto-vectorization - This approach primarily targets loops and depends on dependence analysis to determine legality of the transforms that would generate vector-code that is faster but semantically equivalent to the scalar version. Classical Loop-based auto-vectorization strategies are described in [6], while approaches described in [15,14] are relatively recent advances in auto-vectorization of loops.
2. Straight-line code auto-vectorization - These include SLP-style algorithms, e.g. [8,25,23]. They identify sets of scalar instructions and replace them with vector instructions. These algorithms operate on any straight-line code, anywhere within the program including loop-bodies after loop-optimizations are unable to vectorize the code.

2.1 SLP Vectorization

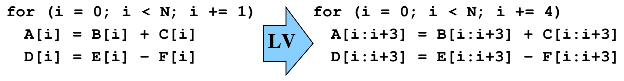
Since PostSLP is inspired by SLP, we will provide a high-level overview of SLP in this section. SLP (Superword Level Parallelism) performs straight-line code vectorization. It does not require a loop structure, instead it can analyze any straight-line piece of code including loop bodies. It collects instructions that can be grouped together into vectors and replaces them with the corresponding vector instructions. For example, given the code on the left hand side of Figure 1, SLP will generate the code at the right hand side.



```
A[i]   = B[i]   + C[i]
A[i+1] = B[i+1] + C[i+1]
A[i+2] = B[i+2] + C[i+2]
A[i+3] = B[i+3] + C[i+3]
```

```
A[i:i+3] = B[i:i+3] + C[i:i+3]
```

Fig. 1: SLP vectorization.



```
for (i = 0; i < N; i += 1)
  A[i] = B[i] + C[i]
  D[i] = E[i] - F[i]
```

```
for (i = 0; i < N; i += 4)
  A[i:i+3] = B[i:i+3] + C[i:i+3]
  D[i:i+3] = E[i:i+3] - F[i:i+3]
```

Fig. 2: Loop-vectorization.

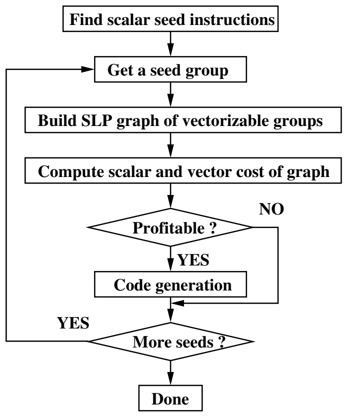

Find scalar seed instructions
Get a seed group
Build SLP graph of vectorizable groups
Compute scalar and vector cost of graph
Profitable ?
NO
YES
Code generation
YES
More seeds ?
Done

Fig. 3: SLP algorithm

The SLP-algorithm begins by identifying set of *seed* instructions. An overview of the algorithm is shown in Figure 3. These sets of instructions are good starting point for vectorization, e.g., stores to consecutive memory addresses, or instructions which form reduction trees. More specifically, the set of *seed*s contains at least 2 instructions, which: 1. have the same bit-width and type; 2. have no dependencies among them; 3. access adjacent locations if they are memory instructions, or form a reduction tree (e.g., a reduction tree of additions). The *seed* instructions are the starting point of the vectorizable graph, i.e., they form the root group node.

Using these *seed* instructions as the root node, the compiler follows the use-def chains towards the definitions, looking for instructions that can form more group nodes. This bottom-up approach of building the graph is followed by most state-of-the-art implementations of SLP-algorithms including LLVM and GCC. When it encounters an instruction that cannot be part of the group, it stops extending the group.

After the SLP graph is built, the algorithm performs profitability analysis of the code, by comparing the cost of leaving the graph in scalar form and cost of vectorizing the instructions in the graph. The analysis should also account for execution of additional instructions that have to be generated. An example would be, if the group generated `<4 x i32>` instructions, but there are external scalar `i32` uses, we have to generate `extractelement` instructions that will move the data from the vectors to the scalars. The accuracy of the target-specific cost-analysis is critical in determining the profitability of the group. Once all the *seed* instructions of the code are exhausted, the process stops.

2.2 Loop Vectorization

Unlike SLP, the loop vectorizer can only operate on loops. It is an inherently different vectorization approach, as it does not work by searching the code for vectorization candidates. Instead, it relies on the semantics of the loop: An instruction in a loop will repeat across the loop iterations. Therefore, the loop vectorizer will widen each instruction in the loop, vectorizing across consecutive loop iterations. Obviously, just like in SLP, vectorization is not always legal, so the loop dependence analysis needs to be queried for the necessary checks. Finally, similarly to SLP, loop vectorization is not always profitable, so the cost model needs to decide on whether the loop should get vectorized or not. In the example of Figure 2, the statement on the left hand side will be vectorized as shown on the right.

3 Motivation

3.1 Restrictive Regions

The example of Figure 4 shows how we can improve the code generated by the state-of-the-art compiler with the help of PostSLP.

Given the code of Figure 4(a), the SLP auto-vectorizer will first notice that the scalar stores `A[i+0]` and `A[i+1]` are consecutive, and the same for the stores to `A[i+3]` and `A[i+4]`. There is a gap between these two group stores, as there is no store to `A[i+2]` in the code. Please note that such code is more commonly a result of struct accesses, rather than array accesses, but we are using the array access notation for better readability of the example.

The SLP vectorizer will form two separate groups out of these two sets of consecutive stores, and will therefore operate on these two seeds independently. Each seed group becomes a separate bottom-up region, as shown in Figure 4(b). Region 1 includes the stores to `A[i+0]` and `A[i+1]` and the rest of the definitions, which is the two additions, the two subtractions and the loads from `B[i+0]`, `B[i+1]`, `C[i+0]`, `C[i+1]` and `D[i+0]`, `D[i+1]`. Similarly, Region 2 includes the stores to `A[i+2]`, `A[i+3]`, and the rest of the definitions, which is the two additions and the two subtractions, and the loads from `B[i+2]`, `B[i+3]`, `C[i+2]`, `C[i+3]` and `D[i+2]`, `D[i+3]`.

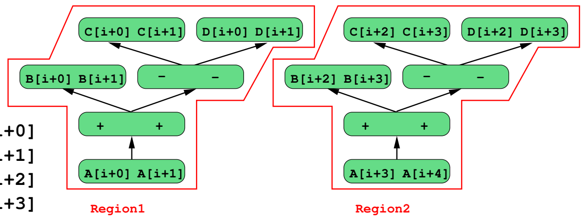
SLP operates on each region independently, as there are no inter-region dependencies that would allow it to cross. SLP's code generation will first generate vector code for Region 1, and will then generate code for Region 2, as shown in Figure 4(c). The first two statements have been vectorized into one 2-wide vector statement, and the two last statements into a second vector statement.

PostSLP can do better than this. If we take a closer look into the code of Figure 4(c), and its corresponding DAG of Figure 4(d), we can see that there is opportunity for further vectorization by combining instructions across regions. The vector loads from `B[i:i+1]` and `B[i+2:i+3]` are accessing adjacent memory locations and can therefore be combined into a more efficient 4-wide vector load from `B[i:i+3]`, as shown in Figure 4(e). Similarly, the loads from `C[i:i+1]` and `C[i+2:i+3]`, the loads from `D[i:i+1]`, and `D[i+2:i+3]` and the 2-wide vector additions and subtractions can all be combined into 4-wide operations. Since the 2-wide vector values are still used by the 2-wide stores to array `A`, we need to add additional `shuffle`[1] instructions that extract the 2-wide subvectors. The resulting wider (partially 4-wide) optimized code is shown in Figure 4(f).

[1] The `shuffle` instructions of these examples are similar to LLVM's `shufflevector` instructions.

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+1] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+4] = B[i+3] + C[i+3] - D[i+3]
```

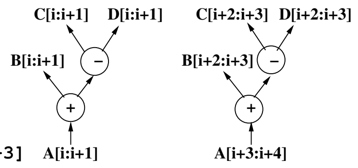
(a) Source Code



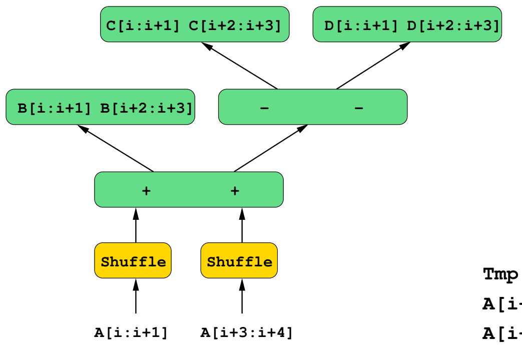
(b) Vectorization Regions

```
A[i+0:i+1] = B[i:i+1] + C[i:i+1] - D[i:i+1]
A[i+3:i+4] = B[i+2:i+3] + C[i+2:i+3] - D[i+2:i+3]
```

(c) After Vectorization, input to PostSLP



(d) DAG before PostSLP



(e) DAG formed by PostSLP

```
Tmp = B[i:i+3] + C[i:i+3] - D[i:i+3]
A[i+0:i+1] = shuffle<0:1>(Tmp)
A[i+3:i+4] = shuffle<2:3>(Tmp)
```

(f) Code generated by PostSLP

Fig. 4: PostSLP properly vectorizes sub-optimal regions.

3.2 Partially Vectorized Code

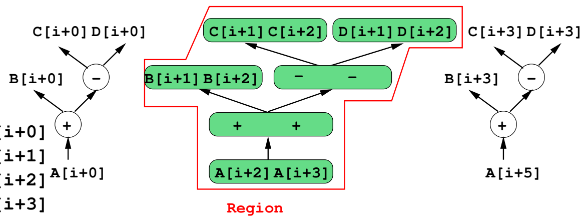
In the previous example of Section 3.1, we showed that the state-of-the-art algorithms can vectorize with smaller than the ideal vector width, because, by design, they operate on regions that restrict the instructions that get considered for vectorization.

In this example we show that the auto-vectorizer may partially vectorize some code, which is an opportunity for PostSLP to further vectorize the code, as shown in Figure 5.

When the state-of-the-art SLP auto-vectorizer is given the code of Figure 5(a), it will identify the two stores to `A[i+2]` and `A[i+3]` as stores to consecutive addresses and will form a seed group out of them. The stores to `A[i]` and `A[i+5]` are accessing memory locations that are not consecutive to the seed group, therefore they are not considered for vectorization. The vectorizer will then from a bottom-up vector region, as shown in Figure 5(b), which includes the two additions, the two subtractions and the loads from `B[i+1]`,

```
long A[], B[], C[], D[]
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2] = B[i+1] + C[i+1] - D[i+1]
A[i+3] = B[i+2] + C[i+2] - D[i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```

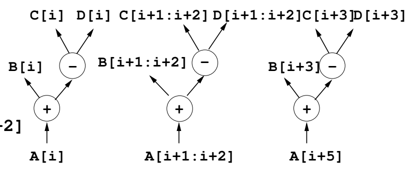
(a) Source Code



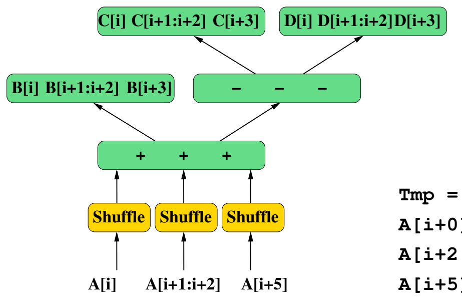
(b) Vectorization Regions

```
A[i+0] = B[i+0] + C[i+0] - D[i+0]
A[i+2:i+3] = B[i+1:i+2] + C[i+1:i+2] - D[i+1:i+2]
A[i+5] = B[i+3] + C[i+3] - D[i+3]
```

(c) After Vectorization, input to PostSLP



(d) DAG before PostSLP



(e) DAG formed by PostSLP

```
Tmp = B[i:i+3] + C[i:i+3] - D[i:i+3]
A[i+0] = shuffle<0>(Tmp)
A[i+2:i+3] = shuffle<1:2>(Tmp)
A[i+5] = shuffle<3>(Tmp)
```

(f) After PostSLP

Fig. 5: PostSLP on partially vectorized code.

B[i+2], C[i+1], C[i+2] and D[i+1], D[i+2]. Since there are no more seeds available to the vectorizer, the rest of the code will remain scalar.

The code generator of the vectorization pass will widen each instruction in the region and will generate the code of Figure 5(c). This code includes two scalar statements A[i+0]=B[i+0]+C[i+0]-D[i+0], A[i+5]=B[i+3]+C[i+3]-D[i+3], and the 2-wide: A[i+2:i+3]=B[i+1:i+2]+C[i+1:i+2]-D[i+1:i+2].

With the help of the PostSLP post-vectorization pass, we can do better. PostSLP figures out that the scalar loads from B[i+0], B[i+1:i+2] and B[i+3] are all consecutive in memory, and will optimize them into a single 4-wide load from B[i+0:i+3], as shown in Figure 5(e). Similarly, the loads from C[i+0], C[i+1:i+2] and C[i+3] can be combined into a 4-wide load C[i:i+3], the loads from D[i+0], D[i+1:i+2] and D[i+3] are combined into D[i:i+3], and also the additions and subtractions can also be combined into 4-wide versions. The resulting code (Figure 5(f)) also contains shuffle instructions that extract the scalars or sub-vector elements from the 4-wide vectors.

3.3 Vectorizing for the Widest Data Type

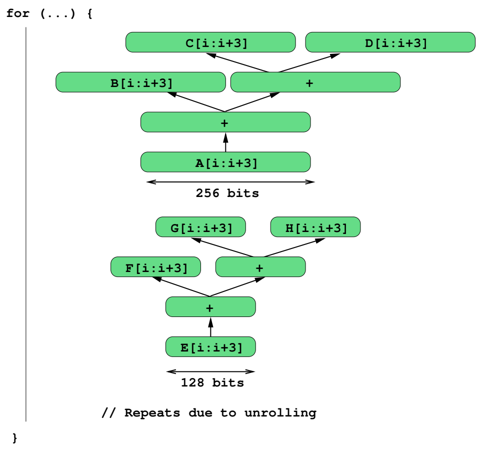
When a loop contains instructions of various bit-widths, for example `f32` (float) and `f64` (double), the vectorizer may try to vectorize based on the widest data type (in this case `f64`). This ensures that the widest vector type corresponds to a legal instruction on the target architecture.

```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; ++i) {
  A[i] = B[i] + C[i] + D[i]
  E[i] = F[i] + G[i] + H[i]
}
```

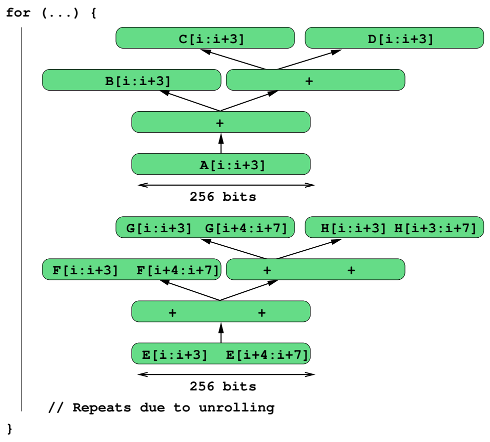
(a) Source Code

```
double A[], B[], C[], D[];
float E[], F[], G[], H[];
for (i = 0; i != N; i += 4 + UF) {
  A[i:i+3] = B[i:i+3] + C[i:i+3] + D[i:i+3]
  E[i:i+3] = F[i:i+3] + G[i:i+3] + H[i:i+3]
  A[i+4:i+7] = B[i+4:i+7] + C[i+4:i+7] + D[i+4:i+7]
  E[i+4:i+7] = F[i+4:i+7] + G[i+4:i+7] + H[i+4:i+7]
  // Repeats due to unrolling UF times
}
```

(b) Vectorized Code



(c) Loop Vectorized



(d) Vectorized by PostSLP

Fig. 6: PostSLP vectorizes wider than the Loop Vectorizer.

Given the code of Figure 6(a), LLVM's loop vectorizer will vectorize it 4-wide for a 256-bit target architecture, because we can only fit 4 instructions of the widest type. LLVM's vectorizer will also unroll the code 8 times, to help increase ILP, leading to the code of Figure 6(b).

Now let's take a closer look into the vector widths generated by the loop vectorizer. Figure 6(c) highlights the vector width differences within the for loop code. Some code trees are 256-bit wide (`A[...] = B[...] + C[...] + D[...]`), while others are 128-bit wide (`E[...] = F[...] + G[...] + H[...]`), leaving the target vector hardware under-utilized. This is where the PostSLP can improve the computational throughput. By grouping together and re-vectorizing the consecutive 128-bit instructions, it can form wider 256-bit instructions, as shown in Figure 6(d). All of the resulting code is 256-bit wide, with (`A[...] = ...`) being 4-wide and (`D[...] = ...`) being 8-wide. This code will obviously perform faster on the target hardware.

4 PostSLP

This section describes the PostSLP algorithm in detail. It is implemented as a separate compiler pass in LLVM's high level optimizer *opt*. It is placed right after the SLP Vectorizer, in order to post-process the code that has been vectorized by either vectorizer. The algorithm resembles the high level structure of the SLP algorithm, but there are several major differences:

1. Its groups can contain both scalars and vectors,
2. It can cross vectorization regions horizontally by considering each widening point as a potential seed,
3. It can grow the graph towards either definitions or uses, similarly to [18],
4. Its code generator and scheduler can effectively generate code for any combination of scalars and vectors in the groups.

4.1 Mix of Scalar and Vector Seeds

The algorithm starts by looking for load and store instructions (either scalars or vectors) that are accessing consecutive memory addresses (Listing 1.1 line 6). All mixed groups of scalar and vector loads or stores that can be combined into wider vector loads or stores with vector sizes supported by the target architecture become the seed groups. For example, the two `<2 x i64>` loads from `B[i:i+1]` and `B[i+2:i+3]` of Figure 4(c) become a valid load group, since a `<4 x i64>` instruction is supported by our target architecture. Similarly, the three loads from `B[i+0]`, `B[i+1:i+2]`, `B[i+3]` of Figure 5(c) are `i64`, `<2 x i64>` and `i64` which can also be combined into a single `<4 x i64>` vector. Please note that unlike LLVM's SLP algorithm, we are not considering a group of scalar-only instructions as a seed, because we are only interested in groups of vectors or groups of scalars and vectors.

Listing 1.1: Main function of PostSLP.

```
1  // Entry point of the function pass.
2  PostSLP(F) {
3    // Go through all basic blocks.
4    for (BB : F) {
5      // Looks for seed instructions.
6      findConsecutiveLoadsAndStores()
7      // The main function of the pass.
8      buildTreesAndCodegen()
9    }
10 }
11
12 // The main body of the algorithm.
13 buildTreesAndCodegen() {
14   while(SeedGroup=getNextSeedGroup()){
15     if (canSchedule(seedGroup)) {
16       // Build tree.
17       growTree(SeedGroup)
18       // Evaluate cost.
19       Cost = getTreeCost(SeedGroup)
20       // Check profitability
21       if (Cost < Threshold) {
22         // Generate code only if
               profitable.
23         codeGenTree(SeedGroup)
24       }
25     }
26     // Remove seed from worklist
27     removeTreeSeeds(SeedGroup)
28   }
29 }
```

4.2 Tree Creation

Then next step is to build the trees for all seeds and generate code (Listing 1.1 line 8). The body of this function is listed in line 13.

The function keeps iterating until it has processed all seeds (line 14). It grabs a seed group and follows both the uses and definitions of each instruction in the group node, in an attempt to grow the tree with new nodes (line 17), similarly to [18]. The body of this function is listed in Listing 1.2 line 2. The candidates for new group node need to meet certain requirements, which are part of the `isCandidate()` function call of Listing 1.2 lines 40 and 19, otherwise they get discarded:

– They have to be of the same opcode, for example all additions, or all stores. Some mixing of opcodes could be allowed with minimal modifications (for example additions with subtractions), but it is currently not supported by our implementation.
– They need to have a single use. This requirement guarantees that we are building a tree, not a generic graph, which simplifies code generation. This constraint could also be removed in a future implementation.
– The type of the instructions (e.g., `int8`, `<4 x float>`, `<2 x i64>` etc.) should: (i) have compatible scalar types, and (ii) should be supported by the target architecture for the desired vector length. For example, a `<5 x i64>` is not supported by a 256-bit AVX2 target.
– The values must not repeat (each instruction is allowed once) and should not be empty.
– The vector size should remain the same across the whole tree. For example, you are not allowed to have both `<4 x i32>` and `<8 x i32>` nodes in the tree.
– The instruction's opcode should be one of the white-listed ones. For example, branches are not vectorized.
– The instruction should not be already part of the tree. If it is, then we don't need to extend the tree further, as these nodes have already been visited. In this case, the code generator will either re-use the vector value of the node, or emit a `shuffle` operation if needed.

In addition to these requirements, it must be legal for the candidates to be scheduled together back-to-back without violating any dependencies. Otherwise vectorizing the instructions would break the program semantics. This is done with the help of an instruction scheduler, being called in Listing 1.2 lines 26 and 26. This is a list-scheduler capable of operating on groups of scalars, vectors, or a mix of the two. It is designed to operate on a dependence DAG that gets built on-the-fly for the instructions within an instruction window that includes all the instructions in the tree so far, including the candidate group.

This whole process of (i) forming a group, (ii) checking its eligibility and legality, and (iii) attempting to grow the tree towards the definitions and uses, repeats until there are no more candidates to add to the tree.

Examples of these trees are shown in Section 3. Figures 4(e) and 5(e) show the group trees formed for the examples of Sections 3.1 and 3.2 respectively. The green nodes represent the instructions that will be grouped together into 4-wide vectors, while the non-colored nodes at the root of the tree show the instructions that will not be modified by PostSLP. Each narrower (scalar or narrower vector) store instruction gets its input through the shuffle nodes (shown in yellow)[2].

[2] In LLVM, we use either the `shufflevector` instructions when the output is a vector instruction, or `extractelement` when the output is scalar

These shuffle instructions create sub-vectors out of the larger input vectors. For example, the the left hand side shuffle of Figure 4(e) reads a `<4 x i64>` input and generates a `<2 x i64>` output that feeds into the `A[i:i+1]` store.

4.3 Profitability

Now that the tree has been completed, we have to check whether vectorizing the tree is profitable (Listing 1.1 line 19) before we generate code. To this end we need to compare the initial cost of the code against the new one, after PostSLP. The cost calculation is done by querying the TTI cost model of LLVM for each individual instruction and summing each individual cost. The profitability function we use considers the following: If the projected cost of the code after PostSLP is less than the original cost (line 21), then PostSLP is considered profitable and we proceed with code generation (line 23). Since PostSLP is capable of handling any mix of scalar and vector instructions, it is particularly important in our case to model the cost of the extraction and insertion from/to vectors or scalars correctly.

4.4 Code Generation (Scheduling and Widening)

The code generation step modifies the IR (Listing 1.1 line 23). It performs two distinct operations.

- The first one performs instruction scheduling on the tree, using the same algorithm as in the tree-building phase (as discussed in Section 4.2). This step will always succeed (i.e, we will not find grows that are illegal to schedule), since we have already checked that each individual group node can be scheduled while building the tree itself (Listing 1.2 lines 26 and 47). This step makes sure that the instructions in each group node are scheduled back-to-back, and are placed in the same order as when the dependencies where checked during `grow_tree()`.
- The second part is instruction widening, which replaces the individual instructions in each group node with their widened counterpart. This is performed in a reverse-post-order traversal of the tree (top-down). A vector instruction is generated with a bit-width equal to the sum of the individual instructions in the group node. For example, if the group contained two `i32` instructions and one `<2 x i32>`, the resulting vector instruction will be of `<4 x i32>` type. The newly generated vector instruction is then emitted in the code and the corresponding group instructions are erased. This new instruction gets its operands from its immediate predecessors in the use-def chains, as they have already been widened by the algorithm, because of the top-down traversal. Finally, for the external uses and operands to the vectorized instructions, the appropriate vector extract/insert instructios get generated using either `insertelement`, `extractelement`, or `shufflevector` instructions.

Once widening is performed, the initial group and any other groups that belong in the group tree get removed from the worklist, as we are done working with it (line 27). Finally, the algorithm repeats until no more groups are left in the worklist (line 14).

Listing 1.2: PostSLP grow tree function

```
// Entry point for growTree function
growTree(SeedGroup) {
 growTreeRec(SeedGroup)
}
// Recursion towards Defs and Uses
growTreeRec(Group) {
 Group.Users = getUsers(Group)
 if (Group.Users)
   growTreeRec(Group.Users)
 Group.Operands = getOperands(Group)
 if (Group.Operands)
   growTreeRec(Group.Operands)
}
// Appends users to Group if possible
getUsers(Group) {
 for (Instr in Group.getInstrs()){
   for (User in Instr.getUsers()) {
     OperandIdx = getOperandIdx(User,
          Instr)
     if (! isCandidate(User)) {
       Group.setMustScatterUsers()
       return
     }
   }
   Users.push_back(User);
   NewGroup = createNewGroup(Users)
   if (! tryScheduleGroup(NewGroup)) {
     delete NewGroup
     Group.setMustScatterUsers()
     return
   }
   Group.setUser(NewGroup)
   NewGroup.setOperand(OperandIdx,
        Group)
 }
}
// Try append operands to Group
getOperands(Group) {
 for (Instr in Group.getInstrs()) {
   for (Oprnd in Instr.getOperands()){
     OprndIdx = getOperandIdx(Instr,
          Oprnd)
     if (! isCandidate(Oprnd)) {
       Group.setMustGatherOperand(
            OprndIdx)
       return
     }
     Operands.push_back(Oprnd)
   }
   NewGroup = createNewGroup(Operands)
   if (! tryScheduleGroup(NewGroup)) {
     delete NewGroup
     Group.setMustGatherOperand(
          OprndIdx)
     continue;
   }
   Group.setOperand(OprndIdx, NewGroup)
   NewGroup.setUser(Group);
 }
}
```

5 Results

5.1 Experimental Setup

We implemented PostSLP in the development branch of LLVM 7 as a separate compiler pass. The pass executes after LLVM's SLP vectorizer pass.We compiled the workloads with the following configurations: *O3:* which corresponds to *-O3* which has all vectorizers enabled by default, and *O3+PostSLP:* which is *-O3* with the PostSLP pass also enabled.

All C/C++ workloads were compiled with `clang/clang++` using *-O3 -march=skylake -mtune=skylake*. The target platform is a Linux-4.9.0, glibc-2.23 based system with an Intel® Core™ i5-6440HQ CPU and 8 GB RAM. We evaluated our approach on unmodified SPEC CPU2006. We also extracted unmodified kernels from SPEC CPU2006 to help focus on code that triggers PostSLP. mFor all results, we report the average of 10 executions, after skipping the first warm-up run. The error bars show the standard deviation. The profitability scores we report are based on LLVM's TTI costs (see Section 4).

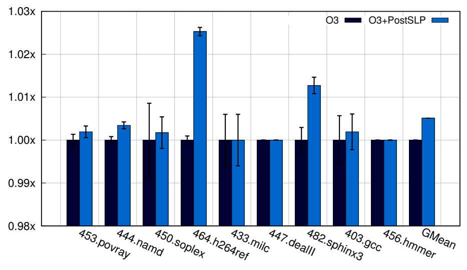
5.2 Performance of Full Benchmarks

We measured the performance of full CPU2006 benchmarks and we report the results in Figure 7(a). The results are normalized to O3. Please note that we are only showing the results for those workloads that triggered PostSLP, the rest of them perform identically to O3. There are three benchmarks that perform significantly better than O3, namely 444.namd (about 0.4%), 464.h264ref (about 2.5%), and 482.sphinx3 (about 1.1%).

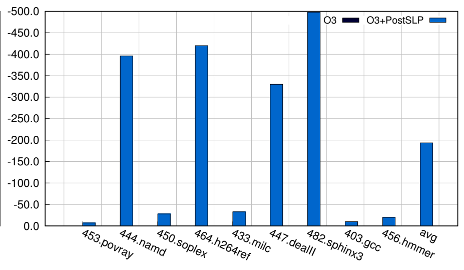
These are very significant performance improvements given that our baseline is an industrial strength compiler, that has been optimized and tuned for

SPEC benchmarks over several years. Moreover, our comparison is a fair one, as PostSLP is designed within the constraints of production compilers: i.e, strict constraints in space and time complexity.

For the remaining workloads, O3+PostSLP performs practically identical to O3, with the performance differences being within the noise margin.



(a) Speedup normalized to O3.



(b) Profitability Score.

Fig. 7: Full benchmark results.

The profitability score is determined statically by the sum of $CostAfter - CostBefore$ for all trees formed by PostSLP. As mentioned in Section 4.3, non profitable trees are discarded by the algorithm and are not considered for code generation. Therefore, the profitability score for any workload is always greater or equal to zero. The more negative the score the better, as negative values correspond to cost savings. The profitability calculation makes use of LLVM's TTI cost modeling API for computing the costs, but as a rule of thumb, a simple fast instruction has a cost of 1.

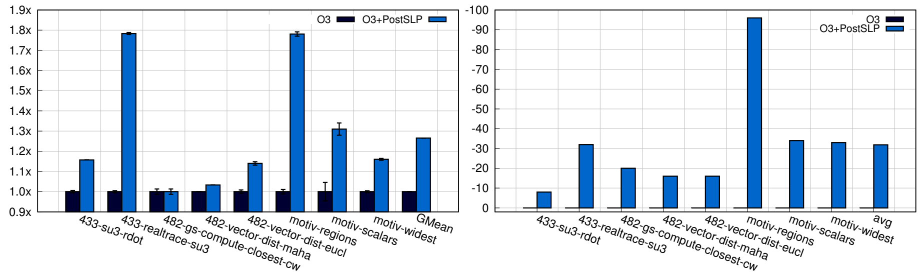
Figure 7(b) reports the accumulated profitability score across each benchmark, as reported by the PostSLP pass. This is a static metric that reports how much more profitable PostSLP was compared to O3. All of the benchmarks that show a significant performance improvement, namely 444.namd, 464.h264ref and 482.sphinx3, also show a large improvement in profitability score. However, other benchmarks like 447.dealII, do not show similar overall speedups, even though their profitability score is quite considerable. This is not a surprising outcome, as the static profitability score does not necessarily correlate strongly with performance improvements. For example, it is possible that all of the profitability improvements correspond to code sections with very small execution time coverage.

5.3 Kernels

To help show that PostSLP has a wider applicability and can help improve real code regardless of its performance impact across a full benchmark, we extracted some functions from CPU 2006 where PostSLP triggered[3] . We reported the execution speedup normalized to O3 in Figure 8(a). We also included the motivating examples of Sections 3.1 and 3.2 for completeness.

[3] We extracted: 433-su3-rdot (su3_rdot.c:10), 433-realtrace-su3 (realtr.c:14) from 433.milc, and 483-gs-compute-closest-cw (gs.c:214), 482-vector-dist-maha (vector.c:266), and 482-vector-dist-eucl (vector.c:238) from 482.sphinx3.

Since these are whole unmodified functions, the performance improvements we are getting depend on the coverage of the code that got widened by PostSLP compared to the rest of the code in the function. Therefore the performance improvements vary considerably across kernels. It is interesting to note that some of the kernels show some very high performance improvements (e.g., the 433.milc derived ones), while the full benchmark they got extracted from, shows no performance difference. This shows that PostSLP does have a wider applicability, regardless of its impact on full spec benchmarks.



(a) Speedup Normalized to O3. (b) Profitability Scores of Kernels.

Fig. 8: Kernel results.

Now, let's consider the profitability scores of Figure 8(b). This plot does indeed show that PostSLP widening improves the profitability of the code, according to the compiler's cost model. However, if we contrast this plot against the results in Figure 8(a), we can see that cost improvements do not necessarily result in actual performance improvements. For example, 482-gs-compute-closest-cw, barely shows any performance improvements, even though it has a healthy profitability score of -20. This is primarily because, in our experience, LLVM's TTI API is overly conservative in reporting costs and is rather oblivious of low-level optimizations which are possible in the back-end.

5.4 Compilation Time

Our PostSLP algorithm introduces a new compilation pass in the pipeline. This introduces a very small compilation time overhead in the general case when compiling large benchmarks. The worst case scenario is when the PostSLP pass gets activated for the main bulk of the workload being compiled. We measured a worst-case compilation-time increase of 14% in a couple of kernels, with a geomean increas of about 8%. Please note that when compiling full benchmarks, the compilation time overhead is barely noticeable.

6 Related Work

6.1 Loop Auto-Vectorization

Auto-vectorization techniques have traditionally focused on vectorizing loops [28]. These loop-based algorithms work by fusing consecutive loop iterations into a single vectorized iteration in a strip-mining fashion, widening each scalar instruction in the loop body to work on vector elements. Early works of Allen and Kennedy on the Parallel Fortran Converter [1,2], the works of Kuck et al. [7], Wolfe [27], and Davies et al. [3] solve many of the fundamental problems of loop vectorizers. Since then, numerous other improvements have been proposed in the literature and implemented in production compilers, e.g. [24,14].

6.2 SLP Auto-Vectorization

SLP Vectorization was first proposed by Larsen and Amarasinghe [8]. It is a complementary technique to loop vectorization which focuses on vectorizing straight-line code instead of loops. Straight-line code vectorization algorithms have been implemented in compilers such as GCC [4] and LLVM, with bottom-up SLP (Rosen et al. [25]) being one that is widely adopted due to its low run-time overhead while still providing good vectorization coverage.

Since its conception, several improvements have been proposed for straight-line-code vectorization in general [26,10,12,5,17]. The widely used bottom-up SLP algorithm has also been improved in several ways [20,19,18,23,22,21]. Combining loop-vectorization and SLP has been explored by [25] and [30]. while [29] enables SIMD widths that are not supported by the target hardware.

6.3 Re-vectorization

Re-vectorization has recently become the focus of research papers that attempt to improve the performance of legacy binaries [11], or legacy hand-vectorized source code [13]. Both approaches implement an SLP-style algorithm along with specific transformations for their domain, e.g., shuffle optimization, unrolling.

PostSLP, on the other hand, is designed to improve auto-vectorization in the common compilation flow of production compilers. To our knowledge, it is the first work that addresses the issues related to vectorization regions and partially vectorized code. PostSLP is a post-processing pass, placed after both vectorizers in the compilation pipeline. It optimizes the sub-optimally vectorized code, by operating across regions, combining both scalars and vectors, further widening instructions that were either completely missed or were sub-optimally vectorized by the production loop and SLP vectorizers.

7 Conclusion

We presented PostSLP, a novel post-processing cross-region vectorization pass that aims at improving the performance of code that has either been partially vectorized, or fully vectorized but with a sub-optimal vector width. It is capable of combining vector instructions, scalar instructions or any mix of the two, generated by either the loop or the SLP vectorization passes. As a result, PostSLP can generate more efficient, wider vector instructions out of smaller vectors or scalars generated by the prior vectorization passes. We have implemented PostSLP in LLVM and have evaluated its effectiveness on SPEC CPU2006. The results show improved performance on real-world code with a minimal increase in compilation time.

References

1. J. R. Allen and K. Kennedy. PFC: A program to convert fortran to parallel form. Technical Report 82-6, Rice University, 1982.
2. J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *TOPLAS*, 1987.
3. J. Davies, et al The KAP/S-1- an advanced source-to-source vectorizer for the S-1 Mark IIa supercomputer. In *ICPP*, 1986.
4. GCC: GNU compiler collection. http://gcc.gnu.org, 2015.
5. J. Huh and J. Tuck. Improving the effectiveness of searching for isomorphic chains in superword level parallelism. In *MICRO*, 2017.
6. K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., 2001.
7. D. J. Kuck, et al Dependence graphs and compiler optimizations. In *POPL*, 1981.
8. S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, 2000.
9. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis transformation. In *CGO*, 2004.
10. J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. A compiler framework for extracting superword level parallelism. In *PLDI*, 2012.
11. Y.-P. Liu, et al Exploiting asymmetric simd register configurations in arm-to-x86 dynamic binary translation. In *PACT*, 2017.
12. C. Mendis and S. Amarasinghe. goSLP: Globally Optimized Superword Level Parallelism Framework. *OOPSLA*, 2018.
13. C. Mendis, A. Jain, P. Jain, and S. Amarasinghe. Revec: Program rejuvenation through revectorization. *CC*, 2019.
14. D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI*, 2006.
15. D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short SIMD architectures. In *PACT*, 2008.
16. OpenMP Application Program Inteface. https://www.openmp.org/specifications/.
17. Y. Park, S. Seo, H. Park, H. Cho, and S. Mahlke. SIMD defragmenter: Efficient ILP realization on data-parallel architectures. In *ASPLOS*, 2012.
18. V. Porpodas. SuperGraph-SLP Auto-Vectorization. In *PACT*, 2017.
19. V. Porpodas and T. M. Jones. Throttling automatic vectorization: When less is more. In *PACT*, 2015.
20. V. Porpodas, et al. PSLP: Padded SLP automatic vectorization. In *CGO*, 2015.
21. V. Porpodas, R. C. Rocha, et al. Super-node SLP: Optimized vectorization for code sequences containing operators and their inverse elements. In *CGO*, 2019.
22. V. Porpodas, R. C. O. Rocha, and L. F. W. Góes. VW-SLP: Auto-vectorization with Adaptive Vector Width. In *PACT*, 2018.
23. V. Porpodas, R. C. O. Rocha, and L. F. W. Góes. Look-ahead SLP: Auto-vectorization in the Presence of Commutative Operations. In *CGO*, 2018.
24. G. Ren, et al. Optimizing data permutations for SIMD devices. In *PLDI*, 2006.
25. I. Rosen, et al. Loop-aware SLP in GCC. In *GCC Developers Summit*, 2007.
26. J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *CGO*, 2005.
27. M. Wolfe. Vector optimization vs. vectorization. In *Supercomputing*. Springer, 1988.
28. M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1995.
29. H. Zhou and J. Xue. A compiler approach for exploiting partial SIMD parallelism. *TACO*, 2016.
30. H. Zhou and J. Xue. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *CGO*, 2016.