Look-Ahead SLP: Auto-vectorization in the Presence of Commutative Operations

Vasileios Porpodas
Intel Corporation, USA
vasileios.porpodas@intel.com

Rodrigo C. O. Rocha
University of Edinburgh, UK
r.rocha@ed.ac.uk

Luís F. W. Góes
PUC Minas, Brazil
lfwgoes@pucminas.br

Abstract

Auto-vectorizing compilers automatically generate vector (SIMD) instructions out of scalar code. The state-of-the-art algorithm for straight-line code vectorization is Superword-Level Parallelism (SLP). In this work we identify a major limitation at the core of the SLP algorithm, in the performance-critical step of collecting the vectorization candidate instructions that form the SLP-graph data structure. SLP lacks global knowledge when building its vectorization graph, which negatively affects its local decisions when it encounters commutative instructions. We propose LSLP, an improved algorithm that can plug-in to existing SLP implementations, and can effectively vectorize code with arbitrarily long chains of commutative operations. LSLP relies on short-depth look-ahead for better-informed local decisions. Our evaluation on a real machine shows that LSLP can significantly improve the performance of real-world code with little compilation-time overhead.

ACM Reference Format:
Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. 2018. Look-Ahead SLP: Auto-vectorization in the Presence of Commutative Operations. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3168807

1 Introduction

Modern compilers include auto-vectorization passes that automatically generate SIMD instructions out of scalar code. Alternatively it is up to the programmer to explicitly expose parallelism with either a vector-aware language, or more commonly with a programming model (e.g. OpenMP [6] pragmas), or even with low-level target-specific intrinsics.

Superword-Level Parallelism (SLP) [25] is the state-of-the-art algorithm for automatically vectorizing straight-line code and has been implemented in several compilers, including GCC [9] and LLVM [14]. It is a bottom-up variant of the original SLP [13] algorithm, for faster compilation time, an important requirement of industrial tools. The algorithm searches through the code looking for vectorizable instruction groups and then generates an equivalent vector code if profitable. It works by first scanning the code for scalars that can become the seeds of vectorization and grouping them together to form the first potentially vectorizable group at the root of the SLP graph. Then, SLP walks up the use-def chains, towards definitions, attempting to group more isomorphic instructions together, as long as they can be potentially vectorized. This process builds the SLP graph, the core data structure of the algorithm. Next, SLP evaluates whether converting the groups of the SLP graph into vectors can improve performance. This cost calculation factors in the overheads of inserting/extracting data into/out of the vector registers. If vectorization is shown to be faster, vector instructions get generated to replace the groups of scalars.

In this work we identify a major limitation at the core of the SLP algorithm, namely, in the formation of the SLP graph, its core data structure. Upon visiting a group of commutative instructions, the algorithm will greedily reorder the operands based mainly on their opcode, with no knowledge of the instructions further up the use-def chains. This reordering is done in an attempt to allow vectorization in cases where the operands of the commutative instructions are vectorizable but just happen to be in the wrong order, as shown in Listing 1. Although this might work well in the

Listing 1. Operands in the wrong order; SLP succeeds.

```
1 load1 = load(A[0])
2 load2 = load(A[1])
3 sub1 = ... - ...
4 sub2 = ... - ...
5 store(E[0]) = sub1 + load1 // incompatible order
6 store(E[1]) = load2 + sub2 // incompatible order
7 // SLP reorders the code to this:
8 // store(E[0]) = sub1 + load1
9 // store(E[1]) = sub2 + load2
```

Listing 2. SLP cannot decide on ordering of operands.

```
1 mul11 = load(A[0]) * load(B[0])
2 mul12 = load(C[0]) * load(D[0])
3 mul21 = load(A[1]) * load(B[1])
4 mul22 = load(C[1]) * load(D[1])
5 store(E[0]) = mul11 + mul12 // SLP can fail
6 store(E[1]) = mul22 + mul21 // SLP can fail
7 // SLP may not perform the required reordering:
8 // store(E[0]) = mul11 + mul12
9 // store(E[1]) = mul21 + mul22
```


CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168807

simplest of cases, the decision is based on a short-sighted view of the code and can lead to early termination of the SLP graph, leading to lower coverage and performance. For example, in Listing 2, the existing SLP algorithm has no way of figuring how it should reorder the operands to guarantee that the code gets vectorized. To make matters worse, the existing reordering fails when the code contains multiple commutative operations of the same type chained together; It will not consider for reordering the operands of the whole chain. Section 3 shows detailed examples that motivate the need for a better operand reordering strategy.

We propose Look-Ahead SLP (LSLP), an algorithm that generates a better SLP graph in the presence of commutative operations. It features: (*i.*) a novel graph construction phase that forms multi-node groups containing chains of commutative operations of the same opcode, and (*ii.*) an improved operand reordering phase that can leverage information from further up the use-def graph for both instructions and memory addresses.

2 Background

2.1 Auto-Vectorization Algorithms

There are two distinct types of auto-vectorization algorithms present in modern industrial compilers:

1. Loop-based algorithms (e.g. [17, 18]) which fuse consecutive loop iterations into a single vectorized iteration in a strip-mining fashion.
2. Straight-line code algorithms, the most common being the fast bottom-up SLP [25], which is inspired by [13]. Their main features are that: (*i.*) they are not restricted to operate within loops, i.e., they can handle straight-line code anywhere in the program, (*ii.*) they can vectorize code within loops where the loop-vectorizer fails.

Both loop-based and straight-line code algorithms conceptually perform the same operation: they reduce VL (Vector-Length) isomorphic instructions into VL-wide vector instructions. However, they follow different approaches to achieve this result. In loop-based algorithms, the presence of the loop structure implies the presence of multiple copies of each instruction in neighboring iterations. Straight-line code algorithms, however, do not rely on the presence of a loop, so they have to scan the code for repeated sequences of isomorphic scalar instructions. A common configuration is to run the SLP pass after the loop-based vectorizer.

2.2 SLP Vectorization

A widely used straight-line code vectorizer is the bottom-up SLP algorithm [25]. Its goal is to find isomorphic instruction sequences and vectorize them if profitable. It works by first scanning the compiler's intermediate representation, identifying specific type of instructions, referred to as *seeds*. The seeds are instructions which are likely to form vector sequences. These are usually stores or instructions that form

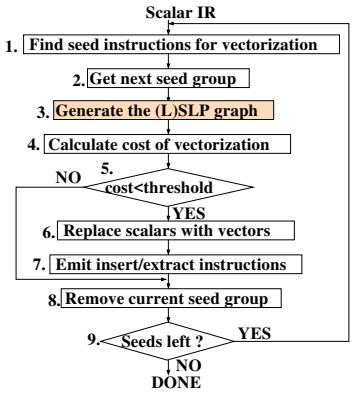


Figure 1. Overview of the bottom-up SLP algorithm.

reduction trees. The seeds become the first potential vector group and are the starting point of the algorithm. The algorithm then searches through the code prior to the seeds, following the use-def chain, to extend the SLP graph in order to form the rest of the vectorizable groups. The code to be vectorized can span multiple basic blocks, as long as each group of instructions to be vectorized belongs to the same basic block.

An overview of the SLP algorithm is shown in the flowchart of Figure 1 (the highlighted section is where LSLP differs from the vanilla SLP algorithm). The SLP algorithm first scans for vectorizable seed instructions (step 1), which are instructions of the same type and bit width that are likely to form vectorized code e.g.: (*i.*) non-dependent store instructions that access adjacent memory locations (scalar evolution analysis [4] is commonly used to test for this), (*ii.*) instructions that lead to idioms such as reduction trees (e.g. a reduction tree of additions), gather-like idioms (e.g., non-consecutive loads), etc. Compilers commonly look for adjacent store seeds first [25], as they are the most promising seeds. The seeds are inserted into a list (step 2).

The algorithm then goes through the seeds and starts to build the SLP graph (step 3). Building the SLP graph involves forming groups of potentially vectorizable instructions by following the data dependence graph that starts at the seed instructions. The state-of-the-art method for generating the graph is to start from store seed instructions and build the graph from the bottom up. This is the approach followed in both GCC's and LLVM's implementations of the SLP vectorizer [25]. Each group contains the scalar instructions that are candidates for vectorization, but it also carries some additional auxiliary data such as the group's cost (see next step). Once the algorithm encounters scalar instructions that cannot form a vectorizable group, it forms a non-vectorizable group which will carry the cost of collecting the data from scalars and inserting them into a vector. At this point the algorithm stops exploring the code in this direction as this path cannot be vectorized any further.

Listing 3. A simplified version of the SLP graph generation.

```
1  build_graph(instrs) {
2   // i. Termination conditions
3   if instrs not vectorizable: return
4   // ii. Append new node to graph
5   graph.add(new_group_node(instrs))
6   // iii. Operand ordering based on opcode
7   reorder_operands(instrs.operands)
8   // iv. Recursion for each operand
9   for operands in instrs.get_operands():
10    build_graph(operands)
11 }
```

After constructing the graph, SLP estimates the performance benefits of the vectorized code (step 4). This is done with the help of the compiler's target-specific cost model. The cost of the graph is equal to the sum of the savings from converting each group of scalar instructions into vector form (the lower the cost the better) plus the overheads for gathering the inputs of the vector instructions. In step 5 the cost of the vectorized code with the SLP graph is compared against a threshold (usually 0) to determine whether code generation of the vector code should proceed (steps 6 and 7). If so, the compiler modifies the intermediate representation code by replacing the groups of scalar instructions with their equivalent vector instructions (step 6), and emits any *insert* or *extract* instructions required for the flow of data between the vector and scalar instructions (step 7). If, however, the cost of vectorization is higher than that of the scalar code, then the code remains unmodified. Afterwards, the current seed group is removed from the list (step 8) and the process repeats for all seeds collected by the SLP front-end (step 9).

2.3 SLP Graph Generation

The SLP Graph is generated in step 3 with the help of the recursive function `build_graph()`, as shown in Listing 3. This function is initially called with the seed instructions as the inputs.

The `build_graph()` recursive function (line 1) has four distinct steps: *(i.)* Check the termination conditions[1] (line 2), *(ii.)* Build the vectorizable nodes and connect them to the rest of the SLP graph (line 4), *(iii.)* Perform operand reordering if it is legal and required (line 6). Only commutative instructions are legal candidates for reordering (not shown for brevity). In vanilla SLP, reordering considers only the opcode of the operands of the current instruction, and in the case of loads whether or not they are consecutive. *(iv.)* Finally, the function calls itself for all operands to continue growing the graph up the use-def chains (line 8).

3 Motivation

This section motivates LSLP with the help of three examples that highlight the weaknesses of the existing SLP algorithm, while demonstrating how LSLP overcomes them.

[1] The instructions must be: i) scalars, ii) isomorphic, iii) unique, iv) all in the same basic block, v) schedulable, and vi) not yet in the SLP graph.

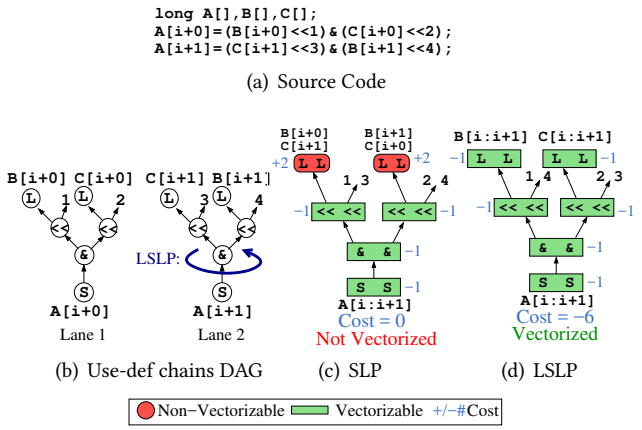


Figure 2. Operand reordering can avoid an unnecessary load address mismatch during vectorization.

3.1 Load Address Mismatch

This example illustrates how a load address mismatch can make vanilla SLP algorithm fail unnecessarily. Figure 2(a) shows the source code that we will use to demonstrate it. The use-def DAGs for this code are shown in Figure 2(b), while SLP is shown in Figure 2(c).

SLP starts from the seeds, which in this example are the consecutive stores to `A[i+0]` and `A[i+1]`. The stores access consecutive memory locations and as such they are vectorizable and are grouped together (green box). Next, SLP follows the data flow up the graph and groups together the pair of bit-wise-and (`&`) instructions. Since the bit-wise-and is a commutative instruction, SLP is free to change the order of the input operands in any way. However, SLP will only do so if the opcodes differ, but in this case the operands are both left-shift («) operations. With no re-ordering taking place, SLP groups the left-shift operations in their original order. Finally, SLP attempts to group the leaf loads `B[i+0]` and `C[i+1]` together into one group, and `C[i+0]` and `B[i+1]` into another. However, these loads do not access consecutive memory locations, therefore all leaf nodes remain as scalars (shown in red).

At this point SLP needs to determine whether it is profitable to generate vector code, given the SLP graph of Figure 2(c). To that end, SLP computes the cost of each node in the SLP graph (integers near the groups of Figure 2(c)). The cost is calculated as the difference VectorCost − ScalarCost, with negative cost values implying better performance of vector code compared to the equivalent scalar code. SLP vectorization cost also accounts for the additional cost of extracting intermediary values with external use, i.e. when they are used by other instructions not in the group of instructions being considered for vectorization. However, for simplicity we assume there is no extraction cost in our examples. The cost of a group is a metric of the overhead of all instructions in the group. A typical integer ALU instruction (e.g., an ADD) has a cost of 1 in both scalar and vector form,

therefore a group cost of −1 (VectorCost = 1, ScalarCost = 2) is quite common for a vectorizable group of two ALU instructions. The actual cost values used result from querying the compiler's cost model[2].

When the operands of a vectorizable group are scalars, these scalar values need to be gathered and aggregated into a vector register, typically leading to the group cost of +2 for the non-vectorizable group in our examples (e.g., Figure 2(c)). If the operand group contains nothing but constants, the gather operation has a cost of zero (as constants vectors and constant scalars can be loaded from memory with equal ease). Mixed groups of constants and scalar instructions get a positive cost proportional to the size of the vector (+2 in our examples, see Figure 3(c)).

In Figure 2(c) the total cost is zero, meaning that vectorization provides no performance benefit, therefore the code remains scalar. LSLP, on the other hand can successfully vectorize the code, as shown in Figure 3(d). LSLP chooses the best reordering of the operands by evaluating a few levels ahead in the tree. The address locations of the load instructions are also considered when reordering the operands of the bit-wise-and (`&`) nodes. The reordering evaluation based on a few levels ahead provides vital insight into how the operands should get reordered for best vectorization, leading to Lane 2 shift-operands getting swapped. The end result is that vectorizable groups are formed for the loads on both sides (`B[i+0]` and `B[i+1]`) and (`C[i+0]` and `C[i+1]`). The total cost is −6, which is profitable for vectorization.

3.2 Opcode Mismatch

This example focuses on a different problem, that of providing information to the vectorizer about the instruction opcodes that are found beyond the currently visited node. Figure 3(a) shows the input code. Its use-def DAGs are shown in Figure 3(b). SLP fails to properly reorder the operands of the addition since both inputs are bit-wise-and (`&`) nodes. As a result, when the algorithm reaches the operands of the bit-wise-and instructions, it fails to form a vectorizable group as they are instructions of different opcode (left-shift («) and addition (+)). The cost of the SLP code is +4, which is non-profitable for vectorization, and as such remains scalar.

In LSLP, on the other hand, the algorithm has access to information about the instructions beyond the current level of the tree. It performs reordering based on the opcodes of the paths leading to the addition. The reordering evaluation dictates that, in Lane 1, the operands of the addition should be swapped, as shown in Figure 3(b). This leads to the SLP graph of Figure 3(d), which is vectorizable up until (excluding)

[2] The compiler's cost model provides a target-dependent cost estimation that approximates the cost of an intermediate representation (IR) instruction when lowered to machine instructions. Our examples make use of the cost values provided by LLVM's target-transformation interface (TTI) for Intel's processor.

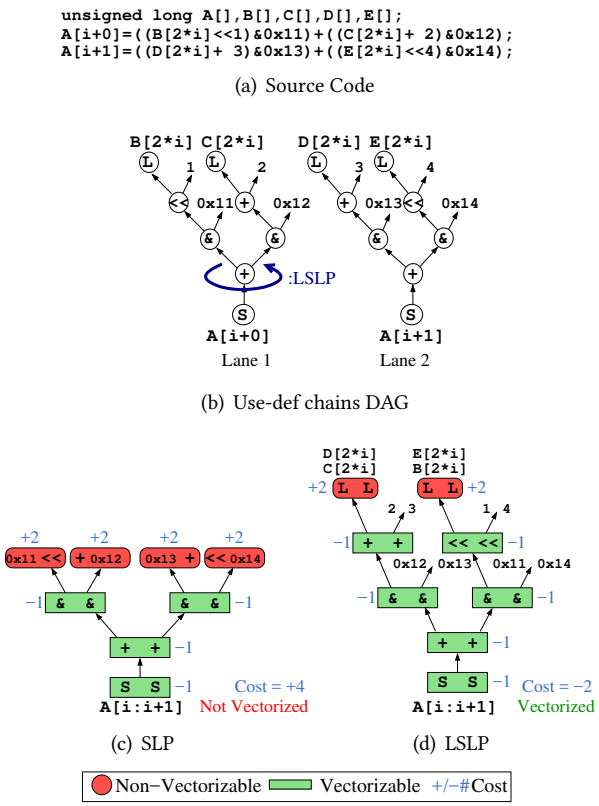


Figure 3. A proper operand reordering is able to avoid an unnecessary opcode mismatch during vectorization.

the leaf nodes. The cost of this SLP graph is −2, which is considered profitable for vectorization.

3.3 Associativity Mismatch and Multi-Nodes

This section motivates the third and final improvement introduced by the LSLP. So far, we have shown that operand reordering considers a single instruction group (for both SLP and LSLP). There are situations, however, where this proves inadequate. Consider, for example, the code in Figure 4(a). The two lines of code contain the same operations but in different evaluation order (associativity), leading to the two visually different use-def DAGs of Figure 4(b).

During the bottom-up traversal of the DAGs, some of the nodes no longer match in all lanes, e.g., the right operands of both bit-wise-and (&) instructions. To complicate things further, none of the previously described techniques of Sections 3.1 and 3.2 can transform the graphs into fully isomorphic. The reason being that the existing SLP can only perform reordering on each node individually.

LSLP can successfully fix the associativity mismatch by:

1. Changing the walking strategy to prioritize consecutive commutative operations of the same opcode,
2. Supporting the formation of large multi-nodes composed of adjacent instructions with equal opcodes (e.g., the consecutive bit-wise-and (`&`)), and

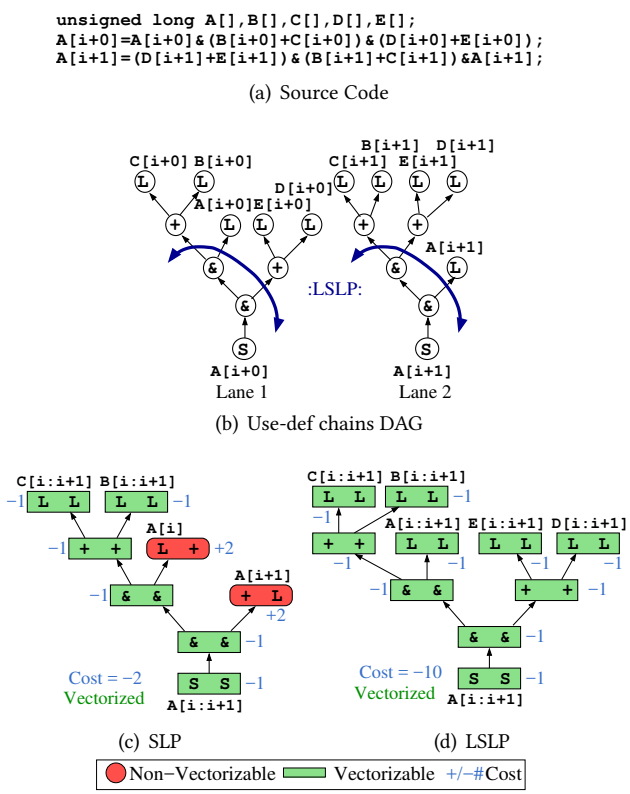


Figure 4. LSLP forms multi-nodes of commutative operations and reorders the multi-node input edges.

3. Performing reordering on all operands of the multi-node using the techniques of Section 3.1 and Section 3.2. The reordering happens at the multi-node frontiers (e.g., across the blue arrows depicted in Figure 4(b)).

Traditional SLP will partially vectorize the code, as shown in Figure 4(c), with a total cost of −2. LSLP, on the other hand, successfully reorders the operands across the edges of the multi-node and generates a fully vectorizable code with a much better cost of −10 (Figure 4(d)).

4 Look-Ahead SLP

4.1 Overview

LSLP introduces several changes at the core of the SLP algorithm: the graph formation (the highlighted step 3 (Generation of (L)SLP graph) of Figure 1). As shown in the examples of Section 3, the graph formation is critical for the effectiveness of the vectorizer as it is the step where the code's isomorphism is explored. The changes introduced by LSLP improve the algorithm's capability of transforming code that is non-isomorphic into equivalent isomorphic code.

In the original SLP algorithm, the generation of the graph follows two states (see Figure 5(a)): (*i.*) Grouping scalar instruction into a group node of vectorizable instructions, and (*ii.*) Reordering the operands.

In LSLP, the graph formation is improved to help extract hidden isomorphism (see Figure 5(b)): (*i.*) It introduces the

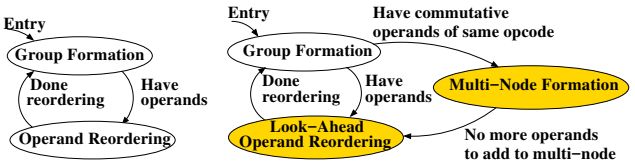


Figure 5. The states of Graph Formation. The highlighted sections are introduced or improved by LSLP.

Listing 4. LSLP Multi-node graph formation

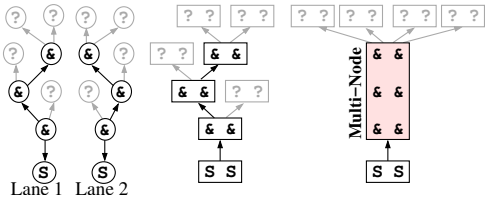
```
1  // In: Array of candidate values for vectorization
2  // Out: Vectorization graph of grouped values
3  build_graph(values) {
4   // Stop growing graph
5   if non-vectorizable values: return
6   // Create new node for values and add to graph
7   graph.add(new_group_node(values))
8   // Recursion call to grow graph further
9   // 1.Commutative
10  if values are commutative:
11    // A. Coarsening Mode
12    for operands in values.get_operands():
13      if (operands' opcode == values' opcode
14          and operands don't escape the multi-node):
15        build_graph(operands)
16      else:
17        multi-node_operands.push_back(operands)
18    // B. Normal Mode: Finished building multi-node
19    if values are the root of multi-node:
20      reorder_operands(multi-node.get_operands())
21      for operands in multi-node.get_operands():
22        build_graph(operands)
23  // 2.Non-Commutative
24  else:
25    for operands in values.get_operands():
26      build_graph(operands)
27 }
```

additional state of Multi-Node Formation, where the graph traversal is redirected to include all chained commutative operands of the same opcode into a single multi-node, before allowing it to proceed further. (*ii.*) The operand reordering is now more powerful, aided by the look-ahead knowledge, and applicable onto multi-nodes. This new reordering allows the algorithm to successfully form more groups, as shown in the examples of Section 3. Both improvements allow LSLP to exploit isomorphism that would otherwise be hidden by the differences in commutativity.

4.2 Multi-Node Formation and Reordering

Traditional SLP performs a simple operand reordering for a single group node of commutative operations. This, however, is not adequate when multiple commutative operations are chained together. LSLP implements a more powerful multi-node reordering while maintaining the efficient bottom-up traversal. This is achieved with a graph building process that works in two modes, within the traditional bottom-up approach. The algorithm is listed in Listing 4.

The `build_graph()` function initially receives an array of *seed* values as arguments (usually consecutive store instructions). The function then proceeds by performing several checks to determine if these values are vectorizable. If



(a) DAG (b) SLP (c) LSLP

Figure 6. LSLP's coarsening step forms a multi-node.

they are not vectorizable, it stops building the graph on this path (line 5). Otherwise, the values are grouped into a single node, which is then attached to the SLP graph (line 7).

The algorithm has now reached the recursion point, which recursively calls `build_graph()` with the operands of the values recently grouped into a vectorizable group node. At this point, there are two modes of operation:

- *Coarsening mode*: When a commutative instruction node is reached, the algorithm switches to *coarsening* mode. This guides the graph formation towards instruction nodes of the same opcode (line 13), eventually leading to the formation of a multi-node composed of several commutative nodes. Since the multi-node's intermediate internal temporary values are not guaranteed to be preserved, the multi-node cannot include values that escape the multi-node (line 14). All the operands of this multi-node are stored into the `multi-node_operands` array (line 17). Figure 6 highlights the outcome of the coarsening mode of LSLP. While SLP forms a group node for each instruction pair (Figure 6(b)), LSLP prioritizes the commutative instruction chain, leading to a large multi-node. Since the multi-node is composed of a chain of commutative instructions with the same opcode, their operands can be re-organized in any order without changing the semantics of the program.
- *Normal mode*: When no more instruction nodes can be added to the coarse node, the algorithm switches back to *normal* mode, continuing its regular process (line 19): Operands are reordered (line 20) and `build_graph` is called recursively for each one of them (line 22).

Finally, non-commutative operations are treated just like in vanilla SLP, where the SLP graph grows following the order of the operands (line 25).

4.3 Top-Level Operand Reordering

The proposed reordering algorithm is a complete redesign of LLVM's reordering algorithm, while maintaining some of its basic features for a fair comparison. LSLP's reordering introduces support for multi-nodes and for the look-ahead exploration and score evaluation. Overall, it performs a single sequential pass over each lane, deciding on the operands order without backtracking [3] (just like the original LLVM algorithm). The top-level function is listed in Listing 5, line 3.

[3]Backtracking can help improve performance, but this study is not in the scope of this paper.

Listing 5. LSLP's Top-level operand reordering

```
1  // Input: Unordered 2D-array (operands x lanes)
2  // Out: Reordered 2D-array (operands x lanes)
3  reorder_operands(operandVec[operand][lane]) {
4   // 1. Strip first lane
5   for i in {0..numOperands}:
6     oper = operandVec[i][0]
7     final_order[i][0] = oper
8     mode[i] = CONST, LOAD or OPCODE depending on oper
9
10  // 2. For all other lanes, find best candidate
11  for lane in {1..lanes}:
12    candidates[] = operandVec[:][lane]
13    // Look for a matching candidate
14    for i in {0..numOperands}:
15      // Skip if we can't vectorize
16      if mode[i] == FAILED:
17        continue
18      last = final_order[i][lane-1]
19      best,mode[i] =get_best(mode[i],last,candidates)
20      // Update output
21      final_order[i][lane] = best
22      // Detect SPLAT mode
23      if (i == 1 and best == last):
24        mode[i] = SPLAT
25  return final_order
26 }
```

Table 1. Brief description of the operand modes.

Mode	Description
`CONST`	look for a constant
`LOAD`	look for a consecutive load to that in the previous lane
`OPCODE`	look for an operation of the same opcode
`SPLAT`	look for the exact same operation
`FAILED`	vectorization has failed, give higher priority to others

The candidates for any lane are all the available operands of that specific lane (line 12). A single commutative instruction has just two operands, but as expected, multi-nodes have more (see Section 4.5).

The first action is to accept the operands of the first lane in their existing order (line 5). This decision is final, and all operand slots (indexes `0` to `numOperands`) are filled in `final_order[i][0]` (line 7). Also, all `mode` slots are initialized (line 8) based on the type of the instruction.

The `mode` array holds the state of each operand slot. It holds information that applies to all lanes but that cannot be inferred by examining only the instruction at the previous lane (we would have to examine multiple instructions). Its purpose is to filter out incompatible nodes in the search for the best candidate for a given slot. In other words, when looking for a candidate, we look for one that matches the current mode, otherwise vectorization is guaranteed to fail for this slot. For example, if the mode is `LOAD`, then unless we find another load instruction, vectorization is guaranteed to fail. If the mode is `FAILED`, it means that vectorization is no longer possible for this operand slot as we could not find a matching candidate in one of the previous lanes. Table 1 shows the list of the modes and their description.

The next step is to go through all lanes from 1 to `lanes`, looking for matching candidates (line 11). We go through all operand indexes (line 14) looking for the best matching candidate, given the current `mode[i]` and the last operand

(line 19). A detailed description on how to select the best operand can be found in Section 4.4 and in Listing 6. The decision is once again final (no backtracking) and the best operand is placed into `final_order[i][lane]`. In case `get_best()` fails to find a good candidate, it will return 'FAILED' along with the default *best* candidate. A failed slot will remain `FAILED` for the remaining lanes. If `best` is exactly the same instruction from the previous lane, then we switch to `SPLAT` mode (line 23) as this improves the vectorization cost.

The more interesting part of the operand reordering process is that LSLP uses the look-ahead technique within the `get_best()` function. It is based on data collected by having the algorithm peeking at the nodes beyond the current level. This is discussed in Section 4.4.

4.4 Finding the Best Operand with Look-Ahead

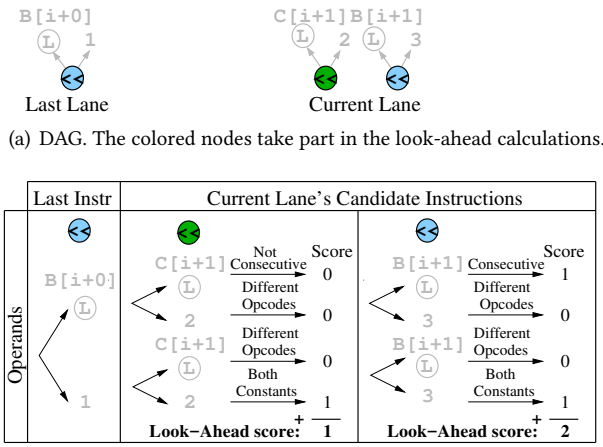
The best operand among the candidates is found either by finding a trivially matching value, or by performing look-ahead. We get a trivial match only if there is a single candidate with matching opcode. If we have multiple matching candidates we break ties by applying the look-ahead technique. The implementation is shown in Listing 6.

As a first step, we collect all matching candidates (that is a consecutive load or operands of the same opcode) into the `best_candidates` array (line 14). For load instructions, the matching test uses scalar evolution analysis in order to compare if two load addresses are consecutive and therefore vectorizable. The matching candidates are then used by both the trivial and the look-ahead parts of the algorithm.

The trivial case is shown in lines 16 to 21. If we have no matches (the `best_candidates` vector is empty), then our search for a vectorizable operand has failed (line 16). Thus, we set the mode to `FAILED` and set the default value (line 11) to be returned. If, on the other hand, we have just a single match in `best_candidates`, then this is the one to return (line 20).

The interesting part of the algorithm is when we have multiple candidates to choose from (line 22). This is a unique feature of LSLP and uses the look-ahead technique. We start from a look-ahead level of 0 and we evaluate the score of each candidate at each level until we either reach the look-ahead limit (line 25), or we manage to break ties and get a clear winner (line 34).

Conceptually, the look-ahead function tries to match the nodes found until a specific depth in the DAG. It tries all possible combinations of the operands of the last value against the operands of the current candidate. The more the nodes that match, the higher the score. As expected, this can become an expensive operation, which is why the maximum level is limited to a small positive integer.



(a) DAG. The colored nodes take part in the look-ahead calculations.

(b) The look-ahead calculations.

Figure 7. Example of the look-ahead calculations.

The calculation of the look-ahead score is defined in Listing 7. Its inputs are: (*i.*) the last lane's value, (*ii.*) the candidate value of the current lane, and (*iii.*) the current look-ahead level. The function recursively calls itself using as arguments all possible combinations of operands of the two values (lines 8 to 9) and sums up the score returned by each recursive call (line 10). When the function reaches either the maximum level (line 5) or values of different kinds (e.g. instruction vs constant), then the function for trivial matching check is used for getting the score (line 6). The sum [4] of all scores collected during the recursion is returned in line 12.

An example of the look-ahead calculation is shown in Figure 7. The last instruction considered is the left-shift («) on the left hand side of Figure 7(a), while the candidates for the current lane are the two left-shifts on the right hand side of Figure 7(a). Nodes of the same color match as their predecessors match. Therefore the look-ahead calculations for those matching nodes should yield a higher score. The calculations are visualized in Figure 7(b). All operands of the last instructions are considered against all operands of the candidate nodes. Each pair of operands contributes to the total score which is the sum of the parts. The light-blue node has the best score of 2, in this example, as both its operands match the operands of the last instruction.

4.5 A Multi-Node Reordering Example

The example of Figure 8 shows how the concepts discussed so far are applied in practice. It visualizes the process of the look-ahead reordering on a multi-node, at the exact moment where the multi-node has formed and its immediate operands are about to be considered for reordering. Figure 8(a) shows the current state of the vectorization graph: the store seed

[4] Alternatively the maximum score could be used instead of the sum. A complete exploration of the heuristics is beyond the scope of this paper.

Listing 6. LSLP's get best candidate

```
1  // Inputs: 1) operand vectorization mode
2  //         2) operand of last lane
3  //         3) array of candidate operands
4  // Outputs: 1) The best candidate (IR value)
5  //          2) The new mode for this operand
6  get_best(mode, last, candidates[]) {
7   switch(mode) {
8   case CONST:
9   case LOAD:
10  case OPCODE:
11    best = candidates[0] // Default value
12    for candidate in candidates:
13      if are_consecutive_or_match(last, candidate):
14        best_candidates.push_back(candidate)
15    // 1. If we have a trivial solution, use it
16    if best_candidates.size() == 0: // No matches
17      mode = FAILED
18      break
19    if best_candidates.size() == 1: // Single match
20      best = best_candidates[0];
21      break
22    // 2. Look-ahead to choose from best_candidates
23    if mode == OPCODE:
24      // Look-ahead on various levels
25      for level in {1..look-ahead-max}:
26        // Best is the candidate with max score
27        for candidate in best_candidates:
28          // Get the Look-Ahead score
29          score = getLAScore(last, candidate, level)
30          if score > bestScore:
31            best = candidate
32            bestScore = score
33        // If found best at level don't go deeper
34        if best and not all scores equal: break
35    break
36  case SPLAT:
37    // Look for other splat candidates
38    for value in candidates:
39      if value == last:
40        best = value; break
41    break
42  case FAILED:
43    // Don't select now, let others choose first
44    best = NULL; break
45  }
46  remove best from candidates[]
47  return (best, mode)
48 }
```

Listing 7. LSLP: Get Look-Ahead Score

```
1  // Inputs: Values to evaluate: val1 and val2
2  //         Maximum Look-Ahead level to explore
3  // Output: The Look-Ahead score (integer)
4  int getLAScore(val1, val2, max_level) {
5   if max_level == 0 or val1,val2 not matching:
6     return (int)are_consecutive_or_match(val1, val2)
7   score_sum = 0
8   for val1_op in val1.get_operands():
9     for val2_op in val2.get_operands():
10      score=getLAScore(val1_op, val2_op, max_level-1)
11      score_sum += score
12  return score_sum
13 }
```

instructions have already formed a group node and the consecutive bit-wise-and (&) instructions have formed a multi-node (highlighted in pink). This is a snapshot of the graph just before the operands of the multi-node are considered for reordering and for creating group nodes out of them for the SLP-graph. The rest of the DAG nodes are shown in faded gray; these nodes are only considered by the look-ahead score calculation. The current state is therefore just before the call to `reorder_operands()` (Listing 4, line 20).

The table of Figure 8(b) summarizes the states of the function `reorder_operands()`, defined in Listing 5, as it decides on the operand ordering from lanes 0 to 3. It tracks the values of the `final_order` 2D-array, the `mode` array, and finally the scores returned by `getLAscore()` (Listing 7) for the candidates being considered. Reordering is performed in a single pass from left to right (lane 0 to lane 4) with no backtracking.

As mentioned in Section 4.3, the first step is to strip the first lane. This sets the operands in the slots in their original order [(«), (L), 1, («)] and also sets the modes to `OPCODE`, `LOAD`, `CONST`, and `OPCODE`, respectively. We are now done with Lane 0 and can proceed to Lane 1.

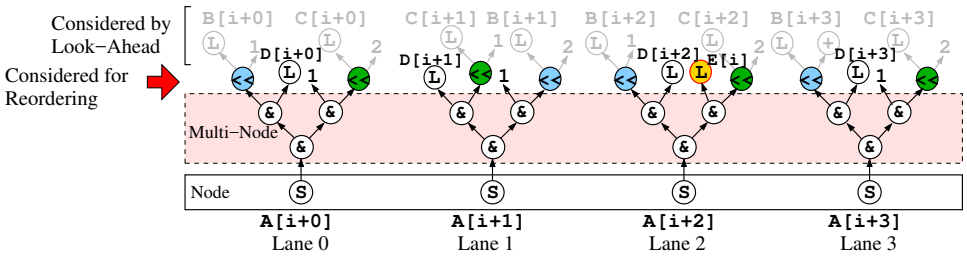
In Lane 1, the candidate nodes are: {(L), («), 1, («)}. It is trivial to fill in slots 1 and 2 as there is only a single candidate that matches the mode of each slot. Slots 0 and 3 require the use of the look-ahead technique to choose between the two possible left-shift («) nodes. The score of each candidate is shown in the table (for slot 0, the light-blue («) has a score of 2 while the green («) has a score of 1). The score, as computed by Listing 7, is the number of matches between the sub-graphs, from the current level onwards. The candidate with the best score is chosen and inserted into the `final_order` array.

In Lane 2, the process repeats. It is important to note that the candidate operands are different than before: there is a load instead of a constant (yellow (L) node loading from `E[i]`). This causes the mode for slot 2 to switch to the `FAILED` state, as it fails to collect a constant that it was hoping for since Lane 0.

Finally, in Lane 3, the operands beyond the left-shift («) have changed. There is an add (+) instead of a constant on its right operand. This causes the look-ahead cost to change and not to favor either of the two shifts. In this case, the algorithm is lucky enough to choose the first of the two, but this is not guaranteed.

The outcome of operand reordering is the final state of the `final_order` 2D-array for all operand slots across all lanes. The light-blue left-shifts are all mapped to slot 0, which will form a vectorizable group node in the SLP graph. All the loads in slot 1 have consecutive accesses to `D[i:i+3]` and will form another vectorizable group. Slot 2 contains three constants and one load, and will therefore not be vectorized. Finally, slot 3 contains the green left-shifts, which will also form a vectorizable group.

The effectiveness of the look-ahead scheme is visible once the algorithm visits the faded nodes, beyond the immediate predecessors of the multi-node. The loads from `C[i:i+3]` (immediate predecessors of the green left-shifts) will end up being vectorized, while the vanilla SLP would fail to vectorize them. The same holds for the loads from `B[i:i+3]`, right before the light-blue left-shifts.



(a) Building the LSLP graph: The immediate operands of the Multi-Node are about to be reordered. Shaded nodes are only considered by look-ahead.

		Lane 0		Lane 1			Lane 2			Lane 3		
		final_order	mode	Look−Ahead score	final_order	mode	Look−Ahead score	final_order	mode	Look−Ahead score	final_order	mode
Operand Slots	0	<>	OPCODE	<>:2 <>:1	<>	OPCODE	<>:2 <>:1	<>	OPCODE	<>:1 <>:1	<>	OPCODE
	1	L	LOAD	N/A	L	LOAD	N/A	L	LOAD	N/A	L	LOAD
	2	1	CONST	N/A	1	CONST	N/A	L	FAILED	N/A	1	FAILED
	3	<>	OPCODE	<>:1 <>:2	<>	OPCODE	<>:1 <>:2	<>	OPCODE	<>:0 <>:2	<>	OPCODE

(b) The internal states and look-ahead scores (the higher the better) that guide the Multi-Node reordering.

Figure 8. An illustrative example that shows how the operand reordering works with multi-nodes and Look-Ahead scores. The internal states are shown in the table. The reordering happens sequentially from Lane 0 to Lane 3, without backtracking.

5 Results

We implemented LSLP in LLVM 4.0 as an extension to the existing SLP vectorizer . We compiled four configurations: (*i.*) O3 which corresponds to *-O3* with all vectorizers disabled, (*ii.*) SLP-NR (No Rotation) which is *-O3* with only the SLP vectorizer enabled but with the operand reordering disabled, (*iii.*) SLP which is *-O3* with only the SLP vectorizer enabled and operand reordering enabled (this is the vanilla SLP), and (*iv.*) LSLP which is *-O3* with only the LSLP algorithm enabled instead of SLP.

All configurations were compiled with `clang` using the following options: *-O3 -ffast-math -mavx2 -march=skylake -mtune=skylake*, and with the loop vectorizer disabled. The target platform is a Linux-4.9.0, glibc-2.23 system with an Intel Core i5-6440HQ Skylake CPU and 8 GB RAM. We evaluated our approach on real code extracted from C/C++ SPEC CPU2006[28] as shown in Table 2, and we included the motivating examples of Section 3 in these tests for completeness. The fortran benchmarks were compiled with the GCC fortran front-end with the help of the DragonEgg [1] project. For all performance and compilation-time results, we report the average of 10 executions, after skipping one initial run. The error bars show the standard deviation. The static cost we report is LLVM's TTI-based cost (see Section 2.2).

5.1 Performance

We measured the execution time of all O3, SLP-NR, SLP and LSLP. The speedup over O3 is shown in Figure 9 and the total static cost seen by each vectorization scheme is shown in Figure 10. A better vectorization algorithm should improve the cost, and if the vectorized code is in a hot part of the code, it should also lead to better performance. We use a maximum of 8 levels for look-ahead in the LSLP reordering.

Table 2. Brief description of the kernels used for evaluation.

Kernel	Benchmark	Filename:Line
453.boy-surface	SPEC2006[28] 453.povray	fnintern.cpp:355
453.intersect-quadratic	SPEC2006[28] 453.povray	poly.cpp:813
453.calc-z3	SPEC2006[28] 453.povray	quatern.cpp:433
453.vsumsqr	SPEC2006[28] 453.povray	vector.h:362
453.hreciprocal	SPEC2006[28] 453.povray	hcmplx.cpp:113
453.mesh1	SPEC2006[28] 453.povray	fnintern.cpp:759
433.mult-su2	SPEC2006[28] 433.milc	m_su2_mat_vec_a.c:23
453.quartic-cylinder	SPEC2006[28] 453.povray	fnintern.cpp:924
motivation-loads	Section 3.1	Figure 2
motivation-opcodes	Section 3.2	Figure 3
motivation-multi	Section 3.3	Figure 4

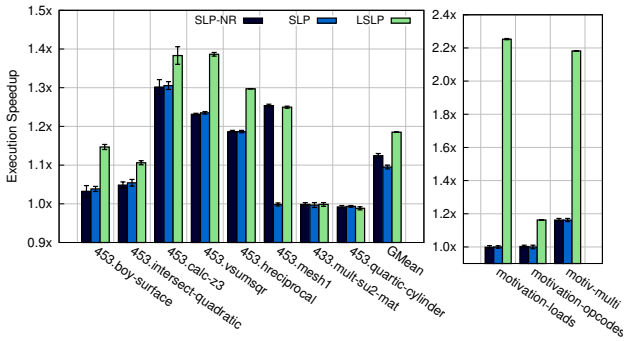


Figure 9. Speedup of LSLP, SLP and SLP-NR over O3.

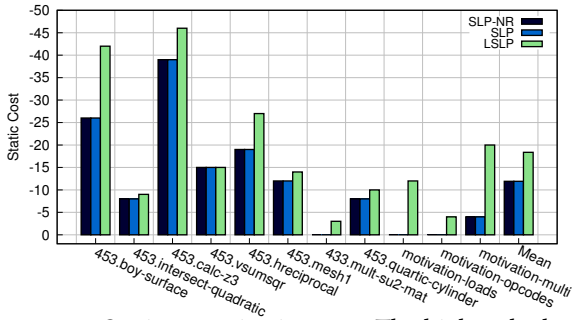


Figure 10. Static vectorization cost. The higher the better.

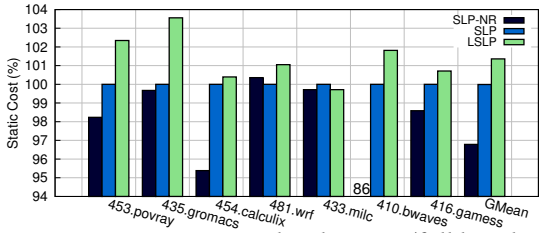


Figure 11. Static cost normalized to SLP (full benchmarks).

The SLP-NR result is of particular importance. It shows that the default rotation heuristic used by vanilla SLP (that is using the immediate predecessor's opcodes to perform the rotation) is not adequate. In these workloads, vanilla SLP achieves about the same performance and static cost as the SLP-NR. Although this result is not representative for the comparison between SLP and SLP-NR, it shows that under difficult conditions where SLP would require look-ahead knowledge or multi-node support, SLP's simple heuristic performs poorly, about the same as SLP-NR.

As it is common in vectorization studies, the performance numbers are not always consistent with the costs reported by the algorithm. Workloads 433.mult-su2-mat and 453.quartic-cylinder are perfect examples of a cost model - performance inconsistency. Although the cost model is showing profitability, the performance achieved is actually worse than O3. Similarly, 453.mesh1 SLP-NR and SLP show the exact same cost, but the generated code and their performance is different, with SLP-NR being superior to SLP. These are all cost modeling issues that require fine-tuning, and not a limitation of the vectorization algorithm itself. The vectorization algorithm relies on the compiler's cost model for checking profitability, which can cause performance regressions if the cost model is inaccurate. The rest of the results show the expected behavior: better costs lead to better performance.

The 453.vsumsqr workload is also rather interesting. Even though the LSLP cost is exactly the same as that of SLP, its performance is significantly better. We examined the generated code and noticed that even though LSLP does reorder the instructions in the graph to make sure that the leaf loads access consecutive memory locations, the *loads* themselves are not being vectorized because there are only three of them, instead of four. However, the code generator identifies this optimization opportunity and performs local packing of some of the consecutive leaf loads for LSLP. This same optimization does not happen with the vanilla SLP as the loads are not on consecutive locations. This is yet another case where the cost modeling proved rather weak.

We also measured the impact of LSLP on whole benchmarks. Figure 11 shows the total cost improvement against the state-of-the-art SLP heuristic and the lack of a reordering heuristic (SLP-NR). Please note that we only show the benchmarks that trigger LSLP, the rest of them behave similarly to SLP. A reordering heuristic is usually beneficial. However, there are cases where the better heuristics perform worse.

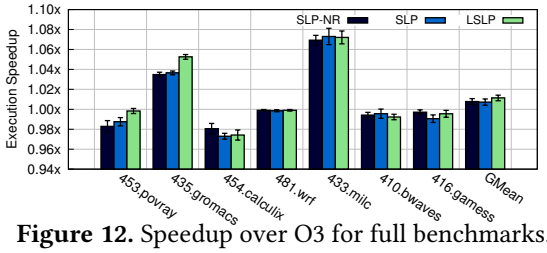


Figure 12. Speedup over O3 for full benchmarks.

For example SLP-NR is better than SLP in 481.wrf and SLP is slightly better than LSLP in 433.milc. However, as we showed in Figures 9 and 10, LSLP can locally improve individual vectorization regions compared to both SLP and SLP-NR. This shows that local heuristics cannot always guarantee a globally better solution. On average, LSLP improves the static cost compared to both SLP and SLP-NR.

We also measured the execution speedup normalized to O3 (Figure 12). The figure shows that the impact of LSLP on whole SPEC benchmarks is rather small. It is about 1% speedup over SLP for 453.povray and 435.gromacs, while the rest are within the noise margin. This is expected behavior, since the regions that get improved by LSLP are not necessarily in hot execution paths.

5.2 Optimizations Sensitivity Analysis

In order to provide insights into LSLP's features that improve performance, we measured each of them in isolation. We measured the following configurations: (i) We started with look-ahead depth of zero (LSLP-LA0) and increased it up to four (LSLP-LA4) while the multi-node size was set to infinity. (ii) We restricted the multi-node size to one (LSLP-Multi1) and incrementally increased it up to three (LSLP-Multi3) while keeping the look-ahead depth at its maximum value of eight. Figure 13 shows the normalized breakdown; the SLP and LSLP bars are shown for reference.

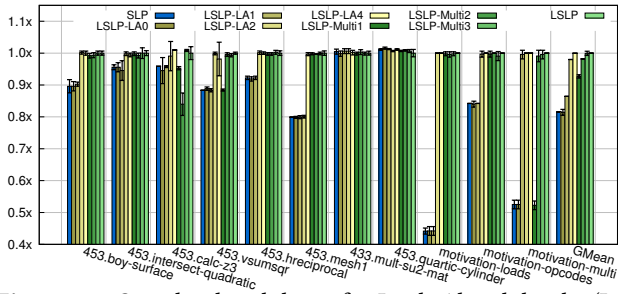


Figure 13. Speedup breakdown for Look-Ahead depths (LA-{1,2,4}) and Multi-Node size (Multi-{1,2,3}).

Overall, as shown in the geo-mean cluster, both optimizations contribute to the performance improvements. A look-ahead depth of four and a multi-node size of three seem to be good values for all our benchmarks. Disabling the look-ahead optimization alone brings LSLP's performance all the way down to SLP's level of performance. This shows that the multi-node formation acts as an enabler for look-ahead to achieve its peak performance.

5.3 Compilation Time

Compilation time is important for industrial compilers. The total compilation wall time normalized to O3 is shown in Figure 14. LSLP is configured with a maximum look-ahead depth of 8. As expected SLP-NR, SLP, and LSLP do increase compilation time over O3 by about 2%, with SLP and SLP-NR being almost identical. On average, LSLP increases the compilation over SLP slightly, by less than 1%, but there are cases where it is both higher (e.g., motivation-{multi, loads}) and lower (e.g., 453.calc-z3).

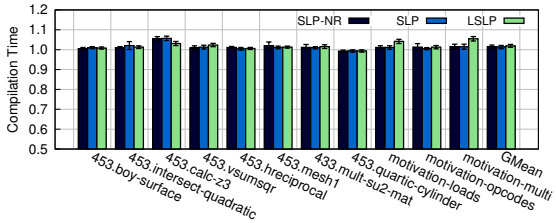


Figure 14. Compilation time, normalized to O3 (LA=8).

One major factor that contributes to vanilla SLP's compilation time is the SLP graph construction. LSLP's look-ahead exploration can make this slower. Occasionally, LSLP may compile faster than SLP (e.g., in 453.calc-z3). This can be attributed to: *(i.)* the formation of fewer, but larger, SLP-graphs, due to its ability to extract more isomorphism, and *(ii.)* reduced instruction count due to more successful vectorization (leading to faster execution of passes that follow).

6 Related Work

High Performance Computing (HPC) has relied on vector machines to accelerate HPC workloads for several decades, while scientific workloads have been accelerated by both commercial [19, 26] and experimental [11, 19, 26] vector machines. General purpose CPUs have also adopted vectorization technology through the use of short SIMD vector instructions [10]. Graphics processors (GPUs) [15] implement similar vector dataphaths for high throughput.

6.1 Loop Auto-Vectorization

Auto-vectorization techniques have traditionally focused on vectorizing loops [30]. The basic implementation conceptually strip-mines the loop by the vector factor and widens each scalar instruction in the body to work on multiple data elements. Many fundamental problems of loop vectorization have been addressed by early work the Parallel Fortran Converter [2, 3] and others [7, 12, 29]. Since then, numerous improvements to the basic algorithm have been proposed in the literature and implemented in production compilers, e.g. [8, 17, 18, 24].

6.2 SLP Auto-Vectorization

The original SLP technique was introduced by Larsen and Amarasinghe [13]. Similar straight-line code algorithms have been implemented in compilers such as GCC [9] and LLVM, with Bottom-Up SLP (Rosen et al. [25]) being widely adopted due to its low run-time overhead and its good coverage. In this paper we use the LLVM implementation of this state-of-the-art SLP algorithm as the baseline.

Since the original SLP work, several improvements have been proposed. Shin et al. [27] propose an SLP-based framework that makes use of predicated execution to successfully vectorize code with control-flow. Barik et al. [5] propose a back-end vectorizer within the instruction selection phase, which makes use of dynamic programming to achieve a superior vector code generation. The Park et al. [20] approach succeeds in reducing the overheads associated with vectorization such as data shuffling and inserting/extracting elements from the vectors. Liu et al. [16] present a vectorization framework that improves SLP by performing a more complete exploration of the instruction selection space while building the SLP graph. Porpodas et al. [23] propose a technique that pads the scalar code with redundant instructions, to convert non-isomorphic instruction sequences into isomorphic ones, thus extending the applicability of SLP. In [22], the SLP region is pruned to scalarize instructions that harm the vectorization cost, while in [21] a larger unified SLP region is used, that overcomes limitations associated with the inter-region communication and unreachable instructions. Finally, Zhou et al. [32] present a vectorization technique that reduces the data re-organization overhead by considering both intra- and inter-loop parallelism, while in [31], they present a technique that enables vectorization of SIMD widths that are not supported by the target hardware.

LSLP is orthogonal to these techniques. It is the first algorithm, that we are aware of, that: *(i.)* exploits the commutative property to generate longer isomorphic instruction sequences and *(ii.)* uses look-ahead information for better selecting the instructions to be included in the SLP graph.

7 Conclusion

We presented LSLP, an improved SLP-based vectorization algorithm that focuses on commutative operations. Firstly, it extends the core of the algorithm and its main graph data structure to form multi-nodes of chains of commutative operations of the same type. Secondly, it introduces a more powerful operand reordering scheme which makes informed decisions based on instructions deeper in the code. Both contributions improve the effectiveness of the algorithm in forming isomorphic instruction sequences, leading to better vectorization coverage and improved performance. LSLP was implemented in LLVM, and improves performance on a real machine with little compilation-time overhead.

Acknowledgments

Rodrigo C. O. Rocha is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant EP/L01503X/1.

References

[1] [n. d.]. DragonEgg: Using LLVM as a GCC backend. http://dragonegg.llvm.org. ([n. d.]).

[2] John R Allen and Ken Kennedy. 1982. *PFC: A program to convert Fortran to parallel form.* Technical Report 82-6. Department of Mathematical Sciences, Rice University.

[3] John Randy Allen and Ken Kennedy. 1987. Automatic Translation of Fortran Programs to Vector Form. *Tranactions on Programming Languages and Systems (TOPLAS)* 9, 4 (1987).

[4] Olaf Bachmann, Paul S Wang, and Eugene V Zima. 1994. Chains of recurrences: A method to expedite the evaluation of closed-form functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC)*.

[5] R. Barik, Jisheng Zhao, and V. Sarkar. 2010. Efficient Selection of Vector Instructions Using Dynamic Programming. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[6] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[7] James Davies, Christopher Huson, Thomas Macke, Bruce Leasure, and Michael Wolfe. 1986. The KAP/S-1- An advanced source-to-source vectorizer for the S-1 Mark IIa supercomputer. In *Proceedings of the International Conference on Parallel Processing*.

[8] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

[9] Free Software Foundation. 2015. GCC: GNU Compiler Collection. http://gcc.gnu.org. (2015).

[10] Intel Corporation. 2007. IA-32 Architectures Optimization Reference Manual. (2007).

[11] Christoforos E Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, R. Thomas, N. Treuhaft, and K. Yelick. 1997. Scalable processors in the billion-transistor era: IRAM. *Computer* 30, 9 (1997).

[12] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Proceedings of the Symposium on Principles of Programming Languages*.

[13] S. Larsen and S. Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

[14] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.

[15] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (2008).

[16] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. 2012. A compiler framework for extracting superword level parallelism. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

[17] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of Interleaved Data for SIMD. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

[18] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization: revisited for short SIMD architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[19] Wilfried Oed. 1992. Cray Y-MP C90: System features and early benchmark results. *Parallel Comput.* 18, 8 (1992).

[20] Y. Park, S. Seo, H. Park, H.K. Cho, and S. Mahlke. 2012. SIMD Defragmenter: Efficient ILP Realization on Data-parallel Architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[21] Vasileios Porpodas. 2017. SuperGraph-SLP Auto-Vectorization. In *2017 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 330–342.

[22] Vasileios Porpodas and Timothy M Jones. 2015. Throttling automatic vectorization: When less is more. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 432–444.

[23] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: Padded SLP Automatic Vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.

[24] Gang Ren, Peng Wu, and David Padua. 2006. Optimizing Data Permutations for SIMD Devices. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

[25] I. Rosen, D. Nuzman, and A. Zaks. 2007. Loop-aware SLP in GCC. In *GCC DevelopersâĂŹ Summit*.

[26] Richard M Russell. 1978. The CRAY-1 computer system. *Commun. ACM* 21, 1 (1978).

[27] J. Shin, M. Hall, and J. Chame. 2005. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.

[28] SPEC. 2014. Standard Performance Evaluation Corp Benchmarks. http://www.spec.org. (2014).

[29] Michael Wolfe. 1988. Vector optimization vs. vectorization. In *Supercomputing*. Springer.

[30] Michael Joseph Wolfe. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley.

[31] Hao Zhou and Jingling Xue. 2016. A compiler approach for exploiting partial SIMD parallelism. *ACM Transactions on Architecture and Code Optimization (TACO)* (2016).

[32] Hao Zhou and Jingling Xue. 2016. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 59–69.