PSLP: Padded SLP Automatic Vectorization

Vasileios Porpodas[†], Alberto Magni[‡], Timothy M. Jones[†]
Computer Laboratory, University of Cambridge[†]
School of Informatics, University of Edinburgh[‡]
vp331@cl.cam.ac.uk, a.magni@sms.ed.ac.uk, tmj32@cl.cam.ac.uk

Abstract

The need to increase performance and power efficiency in modern processors has led to a wide adoption of SIMD vector units. All major vendors support vector instructions and the trend is pushing them to become wider and more powerful. However, writing code that makes efficient use of these units is hard and leads to platform-specific implementations. Compiler-based automatic vectorization is one solution for this problem. In particular the Superword-Level Parallelism (SLP) vectorization algorithm is the primary way to automatically generate vector code starting from straight-line scalar code. SLP is implemented in all major compilers, including GCC and LLVM.

SLP relies on finding sequences of isomorphic instructions to pack together into vectors. However, this hinders the applicability of the algorithm as isomorphic code sequences are not common in practice. In this work we propose a solution to overcome this limitation. We introduce Padded SLP (PSLP), a novel vectorization algorithm that can vectorize code containing non-isomorphic instruction sequences. It injects a near-minimal number of redundant instructions into the code to transform non-isomorphic sequences into isomorphic ones. The padded instruction sequence can then be successfully vectorized. Our experiments show that PSLP improves vectorization coverage across a number of kernels and full benchmarks, decreasing execution time by up to 63%.

1. Introduction

Single Instruction Multiple Data (SIMD) instruction sets as extensions to general purpose ISAs have gained increasing popularity in recent years. The fine-grained data parallelism offered by these vector instructions provides energy efficient and high performance execution for a range of applications from the signal-processing and scientific-computing domains. The effectiveness of vector processing has led all major processor vendors to support vector ISAs and to regularly improve them through the introduction of additional instructions and wider data paths (e.g., 512 bits in the forthcoming AVX-512 from Intel).

Making use of these vector ISAs is non-trivial as it requires the extraction of data-level parallelism from the application that can be mapped to the SIMD units. An automatic vectorization pass within the compiler can help by performing the necessary analysis on the instructions and turning the scalar code to vectors where profitable.

There are two main types of vectorization algorithm. Loop-based algorithms [20, 21] can combine multiple iterations of a loop into a single iteration of vector instructions. However, these require that the loop has well defined induction variables, usually affine, and that all inter- and intra-loop dependences are statically analyzable.

On the other hand, algorithms that target straight-line code [13] operate on repeated sequences of scalar instructions outside a loop. They do not require sophisticated dependence analysis and have more general applicability. However, vectorization is often thwarted when the original scalar code does not contain enough isomorphic instructions to make conversion to vectors profitable.

To address this limitation, we propose Padded SLP, a novel automatic vectorization algorithm that massages scalar code before attempting vectorization to increase the number of isomorphic instructions. The algorithm works by building up data dependence graphs of the instructions it wishes to vectorize. It then identifies nodes within the graphs where standard vectorization would fail and pads each graph with redundant instructions to make them isomorphic, and thus amenable to vectorizing. The end result of our pass is higher vectorization coverage which translates into greater performance.

The rest of this paper is structured as follows. Section 2 gives an overview of a straight-line automatic vectorization technique, showing where opportunities for vectorization are missed. Section 3 then describes our automatic vectorization technique, PSLP. In Section 4 we present our experimental setup before showing the results from running PSLP in Section 5. Section 6 describes prior work related to this paper, and puts our work in context, before Section 7 concludes.

2. Background and Motivation

Automatic vectorization is the process of taking scalar code and converting as much of it to vector format as is possible and profitable, according to some cost model. We first give an overview of the vectorization algorithm that PSLP is based on, then identify missed opportunities for vectorization that PSLP can overcome.

2.1 Straight-Line Code Vectorization

Straight-line code vectorizers, the most well-known of which is the Superword-Level Parallelism algorithm (SLP [13]), identify sequences of scalar instructions that are repeated multiple times, fusing them together into vector instructions. Some implementations are confined to code within a single basic block (BB) but others can follow a single path across multiple BBs, as long as each group of instructions to be vectorized belongs to the same BB. LLVM's SLP vectorizer, and PSLP, follow this latter scheme. The SLP algorithm contains the following steps:

Step 1. Search the code for instructions that could be seeds for vectorization. These are instructions of the same type and bit-width and are either instructions that access adjacent memory locations, instructions that form a reduction or simply instructions with no dependences between them. The most promising seeds are the adjacent memory instructions and therefore they are the first to be looked for in most compilers [28].

Step 2. Follow the data dependence graph (DDG) from the seed instructions, forming groups of vectorizable instructions. It is common for compilers to generate the graph bottom-up, starting at store seed instructions instead of starting at loads. This is the case for both GCC's and LLVM's SLP vectorizers [28]. Traversal stops when encountering scalar instructions that cannot form a vectorizable group.

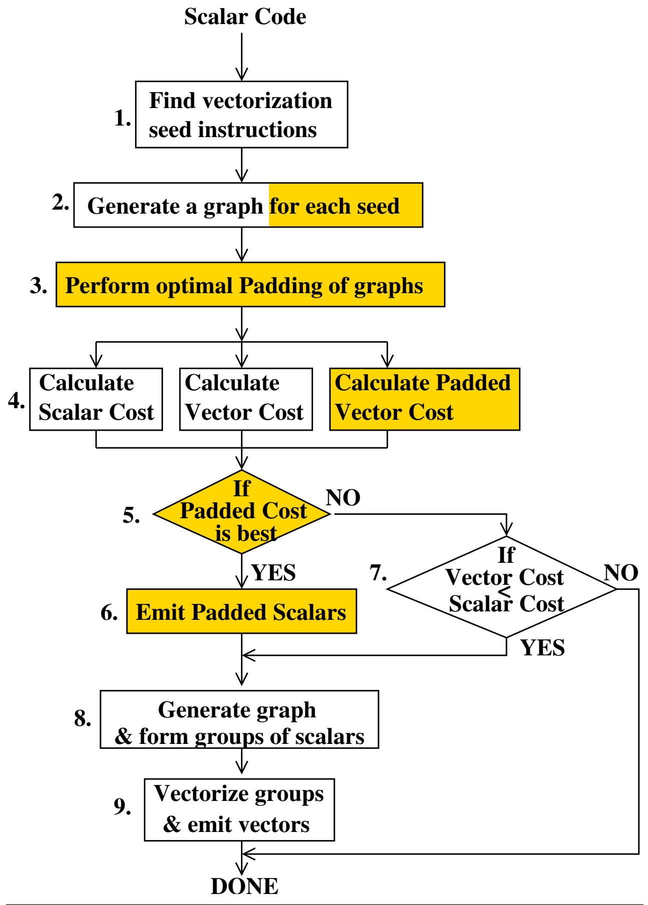
Step 3. Estimate the code's performance for both the original (scalar) and vectorized forms. For an accurate cost calculation the algorithm takes into account any additional instructions required for data movement between scalar and vector units.

Step 4. Compare the calculated costs of the two forms of code.

Step 5. If vectorization is profitable, replace the groups of scalar instructions with the equivalent vector code.

2.2 Missed Opportunities for Vectorization

Although SLP performs well on codes that contain multiple isomorphic sequences of instructions, there are often cases where it cannot actually perform vectorization because the graphs are only similar, not completely the same as each other. These are either directly written by the programmer or, more usually, the result of earlier optimization passes that have removed redundant subexpressions. Figure 2 shows an example and solution to this problem.



Scalar Code
1.
Find vectorization seed instructions
2.
Generate a graph for each seed
3.
Perform optimal Padding of graphs
4.
Calculate Scalar Cost
Calculate Vector Cost
Calculate Padded Vector Cost
5.
If Padded Cost is best
NO
YES
6.
Emit Padded Scalars
7.
If Vector Cost < Scalar Cost
NO
YES
8.
Generate graph & form groups of scalars
9.
Vectorize groups & emit vectors
DONE

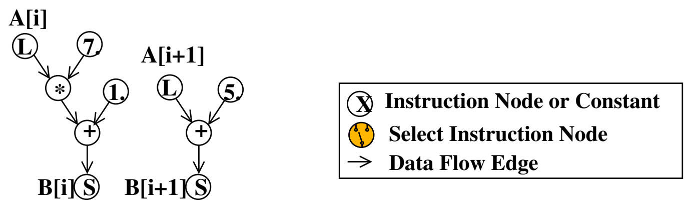
Figure 1. Overview of the PSLP algorithm. The highlighted boxes refer to the structures introduced by PSLP.

In Figure 2(a) we show the original code. The value stored in $B[i]$ is the result of a multiplication of $A[i]$ and then an addition of a constant. The value stored in $B[i+1]$ is only $A[i+1]$ added to a constant.

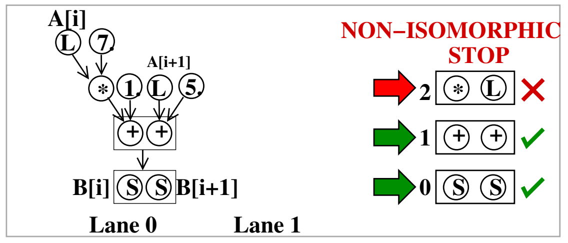
We now consider how SLP optimizes this code, shown in Figure 2(c-d). As described in Section 2.1, we first locate the seed instructions, in this case the stores into $B[i]$ and $B[i+1]$ which are to adjacent memory locations. These form group 0 (the root of the SLP graph in Figure 2(c)). This group is marked as vectorized in Figure 2(d). Next the algorithm follows the data dependences upwards and tries to form more groups from instructions of same type. The second group (group 1), consisting of addition instructions, is formed easily. However, a problem arises when the algorithm tries to form group 2. The available nodes in the graph are a multiplication $(*)$ from the first expression and a load (L) from the second. Since these are not of the same type, vectorization is halted at this point and the algorithm terminates having formed just two groups. Applying the cost model to the two forms of code shows that the packing overheads (associated with inserting the scalar values into the vector registers for the first vector instruction - the addition) outweigh the costs. Therefore this code remains scalar and is compiled down

```
double A[SIZE];
double B[SIZE];
...
B[i] = A[i] * 7.0 + 1.0;
B[i+1] = A[i+1] + 5.0;
...
```

a. Input C code



b. Dependence Graph

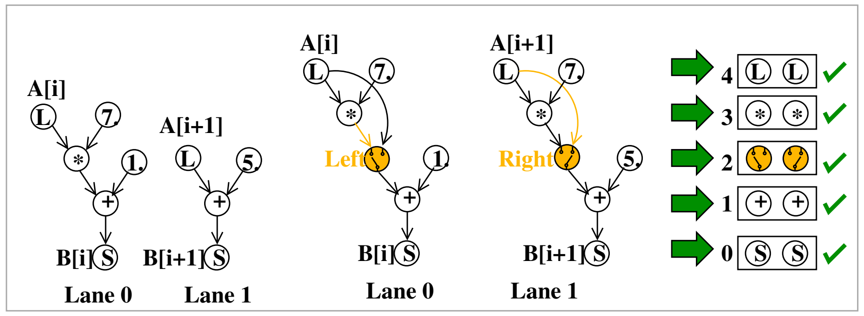


c. SLP internal graph

d. SLP groups

```
vmulsd  A(,%rcx,8), %xmm0, %xmm3
vaddsd  %xmm1, %xmm3, %xmm3
vmovsd  %xmm3, B(,%rcx,8)
vaddsd  A+8(,%rcx,8), %xmm2, %xmm3
vmovsd  %xmm3, B+8(,%rcx,8)
```

e. SLP : Scalar (x86/AVX2 assembly)



```
vmovaps A(,%rcx,8), %xmm2
vmulsd  %xmm0, %xmm2, %xmm2
vaddpd  %xmm1, %xmm2, %xmm2
vmovapd %xmm2, B(,%rcx,8)
```

f. PSLP internal graphs g. PSLP padded graphs h. PSLP groups i. PSLP bypass(x86/AVX2 assembly)

Figure 2. Limitation of state-of-the-art SLP algorithm (c-e) and PSLP solution (f-i). Loads are represented by (L) nodes and stores by (S).

to the x86/AVX2 assembly code[1] shown in Figure 2(e), the same code as is produced with SLP disabled.

The graphs in this example are not isomorphic, but they are similar enough to each other that the smaller graph (in lane 1) is actually a subgraph of the larger. If we were able to match the multiplication from lane 0 (left) with another multiply instruction in lane 1 (right), then vectorization would continue further and would vectorize all the way up to the loads at the leaves of the graph. We can achieve this by padding the smaller graph with redundant instructions—the ones missing from it.

Consider Figure 2(g) where we have performed this padding by copying the multiplication instruction (and its input constant) from lane 0 to lane 1. In addition, to ensure semantic consistency, we have added a new *select* instruction into each graph between the multiplication and addition (orange node marked with the symbol of the electrical switch). In lane 0 the select chooses its left input (the switch points left), which is the result from the multiplication. In lane 1, the select chooses its right input (the switch points right), which uses the result from the load, ignoring the result of the multiplication instruction. This, in effect, makes the multiplication in lane 1 dead code whose result is never used, and it is therefore redundant. However, the overall outcome of this padding is to make the two graphs isomorphic, so that vectorization can succeed.

Figure 2(h) shows how SLP would operate on these two graphs. In total, 5 groups of instructions are created and the whole sequence is fully vectorized, producing the x86/AVX2 assembly code shown in Figure 2(i).

The select instructions get compiled down to vector select operations (like `vpblendv*` in x86/AVX2). In some simple cases, especially in x86/AVX2, they may get compiled down to operations that are semantically equivalent to the selection performed. For example this happens in Figure 2(i) where the vector multiplication along with the vector select (which would filter out the high part of the vector register) have both been replaced by the scalar instruction `vmulsd` (this performs scalar multiplication only on the low part of the vector register). Another possibility is that they could be turned into predicates for the instructions that they select from in case of a target that supports predication and a predication-capable compiler. The select instructions can be

[1] In modern x86 processors scalar floating point operations use the vector register (e.g. %xmm*). In the example all v{mul,add,mov}sd are scalar instructions operating on the first element of the vector %xmm* registers.

further optimized away to reduce under certain conditions, as explained in Section 3.3.4.

3. PSLP

PSLP is an automatic vectorization algorithm that achieves greater coverage compared to existing straight-line code algorithms by padding the data dependence graph with redundant instructions before attempting vectorization. We first give an overview of the algorithm and then describe how the padding is performed.

3.1 Overview

An overview of the PSLP algorithm is shown in Figure 1. The sections that belong to the original SLP algorithm are in white boxes while the PSLP-specific parts are highlighted in orange.

PSLP requires a graph representation of the code for each lane (Step 2), each one rooted at a seed instruction (the graphs are similar to those of Figure 2(f)). These graphs allow PSLP to find padding opportunities and to calculate the minimum number of redundant instructions required to make the graphs isomorphic (Step 3).

In comparison the vanilla SLP algorithm builds a single graph in which each node represents a group of vectorizable instructions, and actually performs vectorization while building the graph. There is no need to build a separate graph for each lane because they would all be identical. The fundamental assumption is that SLP will only vectorize isomorphic parts of the dependence graphs, giving up on the first instruction mismatch and not attempting to fix it.

In PSLP, once the padding has been added to the graph for each lane, we must run the cost model to determine how to proceed. We have to calculate the performance of the original scalar code alongside that of the vectorized code with and without padding (Step 4). If the performance of the padded code (after vectorization) is found to be the best (Step 5), then the algorithm emits the padded scalar code (Step 6). This includes all select instructions and additional redundant instructions that make the graphs isomorphic. If the padded code does not have the best performance, the instruction padding does not proceed and the algorithm decides whether to generate vectors or not, as in vanilla SLP (Step 7). Finally, if vectorization seems beneficial, and regardless of whether padded instructions have been emitted or not, the algorithm will generate the SLP internal graph representation as in Figure 2(c) (Step 8) and will perform vectorization on the groups of scalars (Step 9).

3.2 PSLP Graphs Construction

The PSLP graphs (Figure 1, Step 2) are constructed bottom-up, each of them starting from the seed instructions. For each instruction or constant we create an instruction node and for each data dependence an edge from the definition to the use. An instruction can only be part of a single PSLP graph. The graphs get terminated (i.e., do not grow further upwards) once an instruction is reached that would be shared among graphs. Nodes representing constants are not shared; instead they get replicated such that each graph has its own constants.

3.3 Minimal Graph Padding

At the heart of the PSLP algorithm lies the graph padding function (Step 3 of Figure 1). This function returns a set of isomorphic graphs which have the same semantics as the original graphs but include new select instructions and additional redundant instruction nodes. This ensures that the graphs can be vectorized at a later stage (Steps 8 and 9 of Figure 1).

To obtain optimal performance from the padded code we must find the minimum set of redundant instructions required to get isomorphic graphs. In essence, this is equivalent to finding the minimum common supergraph (MinCS) of a set of graphs. Bunke et al. [5] state that the problem of finding the MinCS of two graphs can be solved by means of maximum common subgraph (MaxCS) computation.

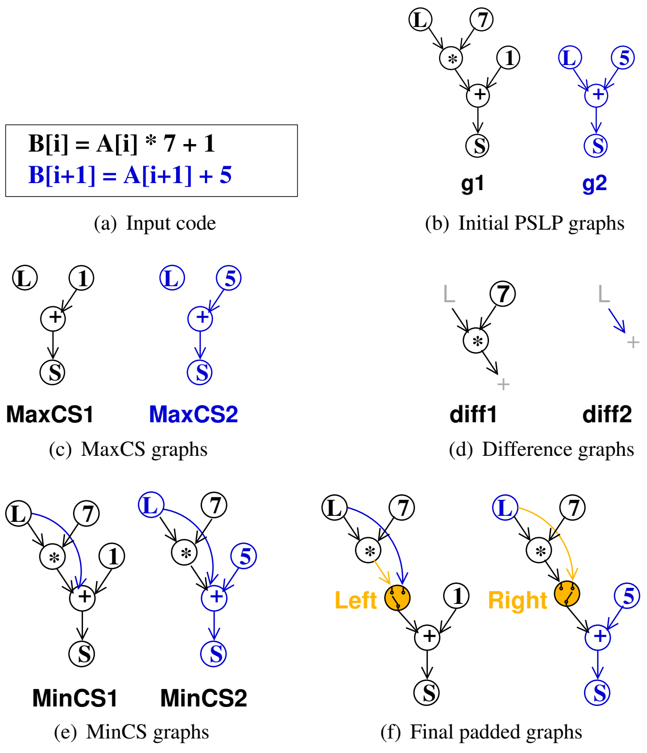
3.3.1 Maximum Common Subgraph

Finding the maximum common subgraph of two graphs $g1$ and $g2$ is an isomorphism problem and is known to be NP-hard. An optimal solution can be achieved with an algorithm such as McGregor's [18], but for any practical usage the graphs should be very small (up to 15 nodes). To address this we propose a faster backtracking algorithm which is very close to the optimal (in this context) and is shown in Algorithm 1. In the example of Figure 3, the input to the MaxCS algorithm are the graphs in 3(b) and the output in 3(c).

Algorithm The two graphs to operate on are given as input to the algorithm. The first thing the algorithm does is to sort the graph nodes so that the graph is traversed bottom-up (line 8), to ensure that children are visited before their parents. This matches the direction of traversal in the later vectorization pass. Next it calls the main recursive function of the algorithm (line 9). The main function starts with iterating over the nodes of $g1$ and attempts to find a node from $g2$ to match with (lines 14 to 33). While searching we skip nodes that:

i. Are already matched (line 17);
ii. Are of incompatible types (line 18);
iii. Will cause cycles in the graph if allowed (line 19);
iv. Are in different basic blocks (line 20); or
v. Cannot be scheduled in parallel (line 21).

Once we find the first two nodes that match we consider them as likely candidates and insert them into a map of nodes from $g1$ to $g2$ and its reverse (line 23 and 24). Then we search for other likely matches within a given radius



(a) Input code (b) Initial PSLP graphs (c) MaxCS graphs (d) Difference graphs (e) MinCS graphs (f) Final padded graphs

Figure 3. The graph padding algorithm.

(line 26) and spawn a new search from that point (line 30). This ensures that we do not get trapped in a local optimal point. These pairs of likely nodes are calculated by the function *get_next_node_pairs()* (line 53). A full-search approach would try out all possible pairs. In our algorithm, however, to keep the complexity low we only consider pairs of nodes that are most likely to lead to a good result. To that end we ignore nodes that have a mobility[2] difference higher than a threshold (line 55). We also apply a cap on the maximum number of pairs that we generate (line 56). The process continues until all nodes pairs are considered by all spawned searches.

When the algorithm exits the loop, the node maps are populated and the algorithm creates two edge maps (mapping common subgraph edges that are in $g1$ to those in $g2$ and vice-versa). This is performed by inspecting each pair of corresponding nodes (lines 35 to 41) and mapping their attached edges one to one. The corresponding edges are the ones that are connected to the corresponding instruction operand of the node. Next, with both node and edge maps we can create the maximum common subgraphs by filtering out nodes and edges that are not present in the maps (lines 43 and 44). Finally, we compare the subgraphs with the best found so far in order to find the maxima (lines 46 to 49).

[2] The mobility of a node in a data dependence graph is calculated as ALAP-ASAP as in [12]

Algorithm 1. Backtracking Maximum Common Subgraph

```
 1 /********* Max Common Subgraph Algorithm *********/
 2 /* Input : Graphs G1 and G2                       */
 3 /* Output: Subgraphs SG1 and SG2                  */
 4 maxcs (g1, g2) {
 5  Initialize node maps: n_map12, n_map21
 6  Initialize the best subgraphs: best_sg1, best_sg2
 7  /* Sort nodes to do the matching Bottom-Up */
 8  Sort g1, g2 nodes by Reverse List Schedule
 9  maxcs_rec (g1,g2,g1.root,g2.root,n_map12,n_map21)
10  return (best_sg1, best_sg2)
11 }
12
13 maxcs_rec (g1,g2,n1_begin,n2_begin,n_map12,n_map21){
14  for (n1 in g1.nodes starting from n1_begin) {
15   for (n2 in g2.nodes starting from n2_begin) {
16    /* Skip incompatible nodes */
17    if (n2 already in n_map21) continue
18    if (n1 and n2 of different types) continue
19    if (n2 is successor of paired nodes) continue
20    if (n1.BB != n2.BB) continue
21    if (can't schedule n1, n2 in parallel) continue
22    /* Found compatible n1 and n2 */
23    n_map12 [n1] = n2
24    n_map21 [n2] = n1
25    /* Look ahead for good node pairs */
26    node_pairs = get_next_node_pairs(n1,n2)
27    node_pairs += get_next_node_pairs(n2,n1)
28    for (pair in node_pairs) {
29     /* Recursion for backtracking */
30     maxcs_rec(g1,g2,pair.n1,pair.n2,n_map12,n_map21)
31    }
32   }
33  }
34  /* Get the edge maps */
35  for (n1 in n_map12) {
36    n2 = n_map12 [n1]
37    e1,e2 = corresponding out edges of n1 and n2
38    /* Set the corresponding edge maps */
39    e_map12 [e1] = e2
40    e_map21 [e2] = e1
41  }
42  /* Generate the subgraphs */
43  sg1 = subgraph(g1,n_map12,n_map21,e_map12,e_map21)
44  sg2 = subgraph(g2,n_map12,n_map21,e_map12,e_map21)
45  /* Keep the best subgraph so far */
46  if (sg1 > best_sg1 and sg2 > best_sg2) {
47    best_sg1 = sg1
48    best_sg2 = sg2
49  }
50
51  /* Return pairs {n1,x} that will likely lead to a
52     good solution. The more the pairs the slower */
53  get_next_node_pairs(n1, n2) {
54    for (n2x in n2.nodes starting at n2) {
55      if (|n2x.mobility - n2.mobility| < m_threshold
56          && pairs.size() < p_threshold)
57       pairs.append ({n1, n2x})
58    }
59    return pairs
60  }
61 }
```

3.3.2 Minimum Common Supergraph

The MaxCS graph is used to calculate the minimum common supergraph. The MinCS is the graph obtained from adding the MaxCS and the differences between the original graphs and the MaxCS graphs [5]. To be more precise, if $g1$ and $g2$ are the original graphs and $MaxCS$ is the maximum common subgraph of $g1$ and $g2$, then $MinCS1 = MaxCS + diff1 + diff2$ where $diff1 = g1 - MaxCS$ and $diff2 = g2 - MaxCS$.

Example Figures 3(a) to 3(e) show an example of calculating the MinCS. The input code and initial graphs are shown in Figures 3(a) and 3(b) respectively. The MaxCS algorithm (Section 3.3.1) gives us the maximum common subgraphs of Figure 3(c). Following the process of Section 3.3.2, we calculate the differences *diff1* and *diff2* (Figure 3(d)) and add them to $MaxCS1$ and $MaxCS2$, resulting in the final MinCS graph of Figure 3(e).

The instructions padded into $MinCS1$ are those in *diff2* and the instruction padded into $MinCS2$ are from *diff1*. Instruction padding involves creating copies of existing instructions and emitting them into the code stream before their consumers. Almost all padded instructions can be directly vectorized along with their corresponding original instructions from the neighboring graphs. The exceptions are load instructions which cannot be vectorized along with identical copies of themselves, since they point to the same memory location—not consecutive addresses. Therefore padded loads also get their addresses modified to point to consecutive addresses.

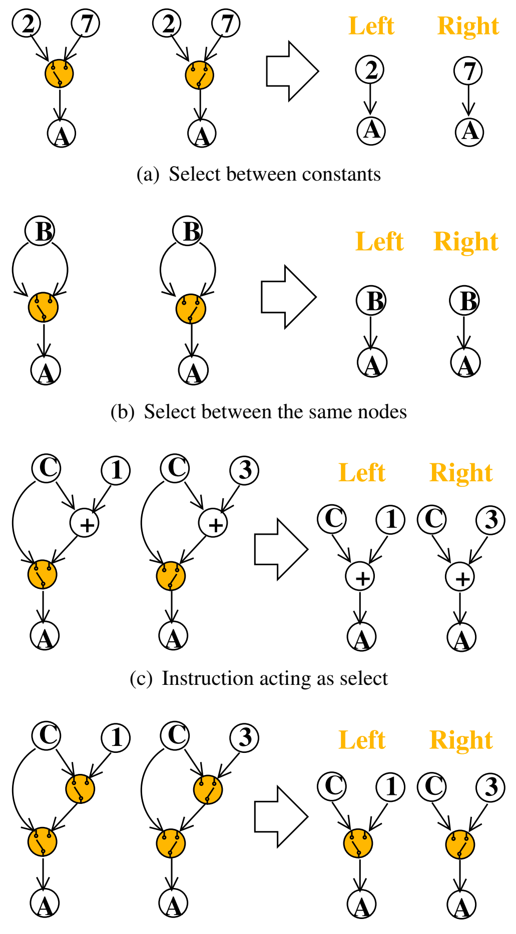
3.3.3 Select Node Insertion

The MinCS graphs do not represent valid code. Node "+" of MinCS1 in Figure 3(e) has 3 input edges, 2 of which flow into its left argument. This is clearly not correct. Therefore PSLP emits select instructions to choose between the edges flowing into the same argument of an instruction node. The selects should always choose the *diff* that belongs to the original code. In this way the semantics of the code do not change with the newly injected code. Given the placement of the *diff* subgraphs on the graph, the selects inserted into the left graph choose the left edge while those inserted into the right graph select the right edge. In Figure 3(f) the left select chooses the multiplication while the right select chooses the load (highlighted edges).

Even though the graphs have been padded with several instructions, it is important to note that the semantics of the computations performed by the graphs is unaffected. The left graph of Figure 3(f) is still $B[i] = A[i] * 7 + 1$ and the right graph is still $B[i+1] = A[i+1] + 5$.

As a subsequent phase in PSLP we could convert the select instructions into predication to achieve the same goal. Predication has two advantages over using select instructions:

1. There is no need to emit and to execute select instructions. This results in fewer instructions and could lead to better performance.
2. It is possible to safely enable/disable instructions with side-effects (e.g., stores) or instructions which could cause an exception (e.g., loads). This may further increase the coverage of PSLP.



(a) Select between constants

(b) Select between the same nodes

(c) Instruction acting as select

(d) Cascaded selects

Figure 4. Optimizing away select nodes.

Our target instruction set is Intel's AVX2 which does not support predicated execution, therefore we leave the evaluation of PSLP with predication for future work.

3.3.4 Removing Redundant Selects

The final stage of graph padding is to remove any select instructions that can be optimized away. This is counter-intuitive, since the MinCS algorithm finds a near-optimal solution. However, MinCS is designed for generic graphs, treating each node equally, whereas we are applying it to PSLP graphs where nodes are instructions or constants and edges represent data-flow. We can use this semantic information to optimize certain select nodes away. Removing a select across all lanes is always beneficial since it will lead to code with fewer instructions.

We have implemented four select-removal optimizations, shown in Figure 4. These remove:

1. Selects choosing between constants through replacement with the selected constant (e.g., Figure 4(a));
2. Select nodes with the same predecessor for each edge (e.g., Figure 4(b));

3. Selects that choose between an instruction that has an identity operation (e.g., *add* 0 or *mul* 1) reading a constant and a node which is common to both the select and that instruction (see Figure 4(c)); and
4. Cascaded select nodes (Figure 4(d)).

The effect of these optimizations is to remove redundant select instructions from the vectorized code, improving its performance.

3.3.5 Multiple Graph MinCS

Until now we have only shown how to derive the MinCS graph from a pair of graphs. However, SIMD vector units have many more than just 2 lanes. In PSLP we compute the MinCS of a sequence of more than 2 graphs in a fast but sub-optimal way, by calculating the MinCS for pairs of graphs and using the computed MinCS as the left graph for the next pair.

For example if we have 3 graphs: $g1$, $g2$ and $g3$ (Figure 5(a)), we compute the $MinCS12$ of $g1$ and $g2$ (Figure 5(b)), and then the $MinCS123$ between $MinCS12\_R$ and $g3$ (Figure 5(c)). In this left-to-right pairwise process we get the rightmost $MinCS123\_R$ supergraph which is the largest of them all[3]. This process introduces redundant selects (for example, the two select nodes in $MinCS123\_R$). These get optimized away by our fourth select removal optimization from Section 3.3.4.

Having obtained the final MinCS, we use this supergraph as a template to recreate the semantics of each of the original graphs[4]. This is performed in two steps. First we map the instructions from each graph to the nodes of this rightmost supergraph and, second, we set the conditions of the select nodes correctly. The end result is from our example is shown in Figure 5(d). After applying select removal optimizations we obtain the padded vectorizable graphs in Figure 5(e).

3.4 Cost Model

Having padded the input graphs, we must make a decision about whether to proceed with vectorization of the padded graphs or the original graphs, or whether to simply keep the code scalar. To do this without degrading performance PSLP requires an accurate cost model for estimating the latency of each case. We largely reuse the cost model of the LLVM SLP pass with minor modifications. The cost is computed as the total sum of all individual instruction costs. Each instruction cost is the execution cost of that instruction on the target processor (which is usually equal to the execution latency). If there is scalar data flowing in or out of vector code, or vector data flowing in or out of the scalar code, then there is an extra cost to be added: the cost of the additional instructions required to perform these data movements.

This cost model is simple to derive, but effective. A more precise model, especially for simpler in-order architectures, would involve instruction scheduling on the target pipeline. The schedule depth would then provide a better estimate of the execution cost. In this way the scalar cost on wide-issue architectures would be estimated more accurately. How-

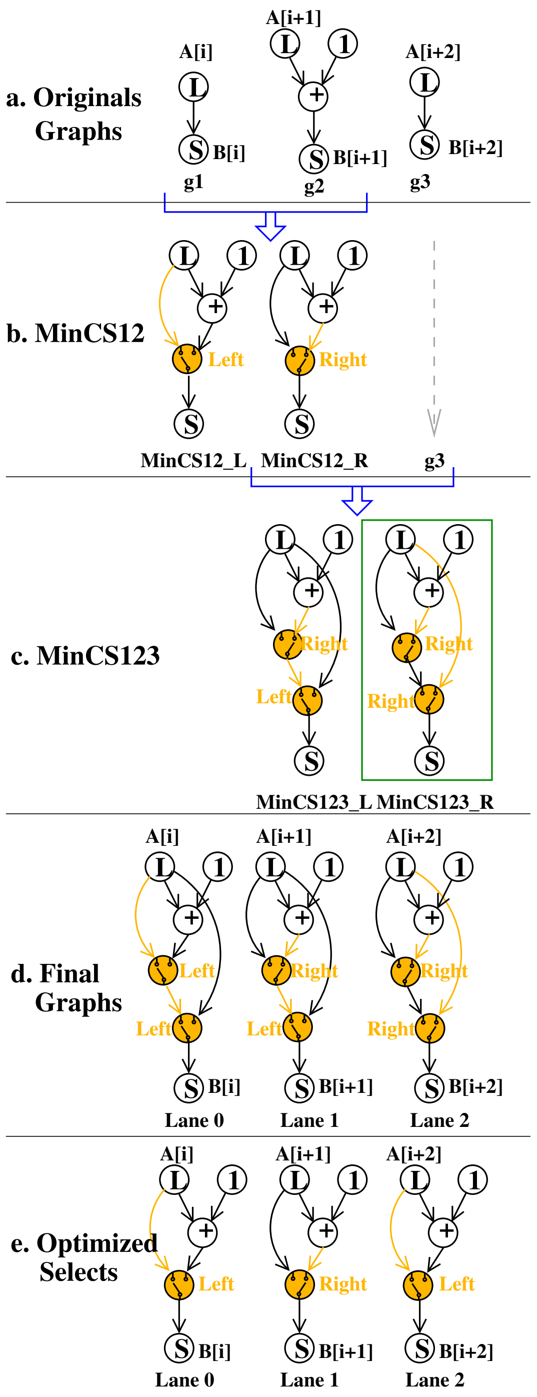


Figure 5. Generation of the MinCS of a sequence of graphs.

[3] This is not always true. In the pair-wise process of generating MinCSs we may discard some nodes leading to a smaller graph.

[4] If the rightmost supergraph is not a superset of all the graphs, it is impossible to guarantee that all nodes of the input graphs can map to the nodes of the final supergraph. We therefore keep graphs acquired only from the left-to-right pairwise calculation of the supergraphs.

Kernel	Description
conjugates	Compute an array of conjugates
su3-adjoint	Compute the adjoint of an SU3 matrix (in 433.milc)
make-ahmat-slow	Take the traceless and anti-hermitian part of an SU3 matrix and compress it (in 433.milc)
jdct-ifast	Discreet Cosine Transform (in cjpeg)
floyd-warshall	Find shortest paths in a weighted graph (Polybench)

Table 1. Description of the kernels.

ever, for our out-of-order target architecture the existing cost model already gives good accuracy; improving it further is beyond the scope of this work.

3.5 Summary

We have presented PSLP, a straight-line code vectorization algorithm that emits redundant instructions before vectorizing. These additional instructions help in transforming the non-isomorphic dependence graphs into isomorphic. The algorithm relies on calculation of the maximum common subgraph to derive the minimum common supergraph which contains the new instructions. Select operations are inserted to keep the program semantics intact. A cost model is then applied to determine how vectorization should proceed. This ensures that enough code is vectorized to offset the overheads from the insertion of additional instructions that PSLP requires (e.g., selects and redundant operations).

4. Experimental Setup

We implemented PSLP in the trunk version of the LLVM 3.6 compiler [14] as an extension to the existing SLP pass. We evaluate PSLP on several kernels extracted from various benchmarks of the SPEC CPU 2006 [31], MediaBench II [7] and Polybench [26] suites. A brief description of them is given in Table 1. We also evaluated PSLP on full benchmarks from the C/C++ benchmarks of SPEC CPU 2006 and MediaBench II suites. We compiled all benchmarks with the following options: *-O3 -allow-partial-unroll -march=core-avx2 -mtune-core-i7 -ffast-math*. The kernels were compiled with more aggressive loop unrolling: *-unroll-threshold=900*. We only show the benchmarks that i) trigger PSLP at least once and ii) show a measurable performance difference due to PSLP; we skip the rest.

The target system was an Intel Core i5-4570 at 3.2GHz with 16GB of RAM and an SSD hard drive, running Linux 3.10.17 and glibc 2.17.

We ran each kernel in a loop for as many iterations as required such that they executed for several hundred milliseconds. We executed the MediaBench II benchmarks 10 times each, skipping the first 2 executions. We executed the SPEC CPU 2006 benchmarks 10 times each, using the reference dataset.

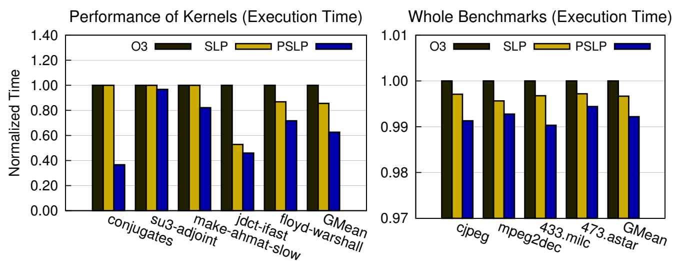


Figure 6. Execution time of our kernels and benchmarks, normalized to O3.

```
b[0].real = a[0].real
b[0].imag = - a[0].imag
b[1].real = a[1].real
b[1].imag = - a[1].imag
```

(a) Non-isomorphic source code (conjugates)

```
tmp1 = quantval[0]*16384
tmp2 = quantval[1]*22725
tmp3 = quantval[2]*21407
tmp4 = quantval[3]*19266
```

opt

```
tmp1 = quantval[0]<<14
tmp2 = quantval[1]*22725
tmp3 = quantval[2]*21407
tmp4 = quantval[3]*19266
```

(b) Non-isomorphism caused by high-level optimizations (jdct-ifast)

Figure 7. Opportunities for PSLP.

5. Results

5.1 Performance

Execution time, normalized to O3, is shown in Figure 6. We measure O3 with all vectorizers disabled (O3), O3 with only SLP enabled (SLP) and O3 with PSLP enabled (PSLP).

The results show that when PSLP is triggered it always improves performance over SLP. This is to be expected as PSLP will generate code only if the cost-model guarantees that its performance is better than both the scalar code (O3) and that vectorized by SLP (see Figure 1, steps 4, 5 and 7).

The kernels *conjugates* and *jdct-ifast* contain two different types of code that benefit from PSLP (see the simplified kernel codes in Figure 7). On one hand, *conjugates* contains a sequence of non-isomorphic computations in its source code (Figure 7(a)), similar to the motivating example in Section 2.2. On the other hand, *jdct-ifast* contains isomorphic computations in the source code which become non-isomorphic after high-level optimizations (Figure 7(b)), in this case strength reduction. The computation in the *jdct-ifast* source code performs a sequence of multiplications against an array of constants. Some of the constant values in the array of constants happen to be a power of two. The multiplication instructions against these specific values get strength-reduced down to logical left shifts, thus breaking the isomorphism. Therefore SLP fails, while PSLP is able to restore the isomorphism and vectorize the code.

In some cases, vectorization can cause performance degradation (not shown in the results). It is not uncommon

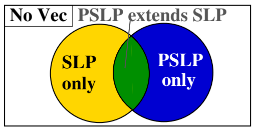


Figure 8. PSLP allows vectorization either when SLP would have failed (shown by "PSLP-only"), or when SLP would have succeed but not to the depth that it does with PSLP (shown by "PSLP extends SLP").

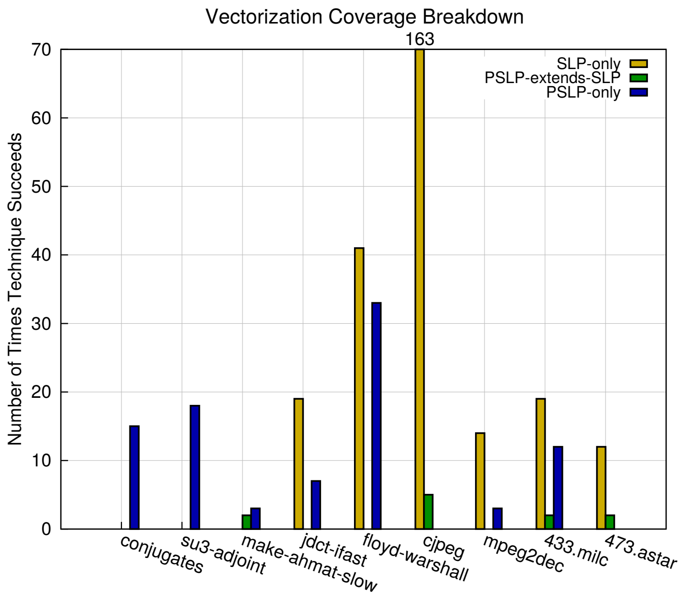


Figure 9. The static coverage of each technique.

for vectorizing techniques to generate code that is slower than the scalar code. This problem can be caused by:

- Low accuracy of the cost model;
- Poor description of the target micro-architecture pipeline in the compiler; or
- The compiler back-end generating very different and slower code from the code at the high-level IR.

So there could be cases, particularly when the target is a powerful wide-issue superscalar processor, where a sequence of scalar instructions is faster than their vectorized counterparts and the cost model could do a bad estimation. This problem, though, does not affect the cost comparison between PSLP and SLP considerably because it is a relative comparison between vector code and vector code. PSLP can avoid the slowdowns compared to SLP, but not compared to scalar code.

5.2 Vectorization Coverage

The main goal of PSLP is to increase the vectorization coverage. More vectorized code usually translates to better performance. To evaluate the coverage, we count the number of times each of the techniques are successful (that is, how

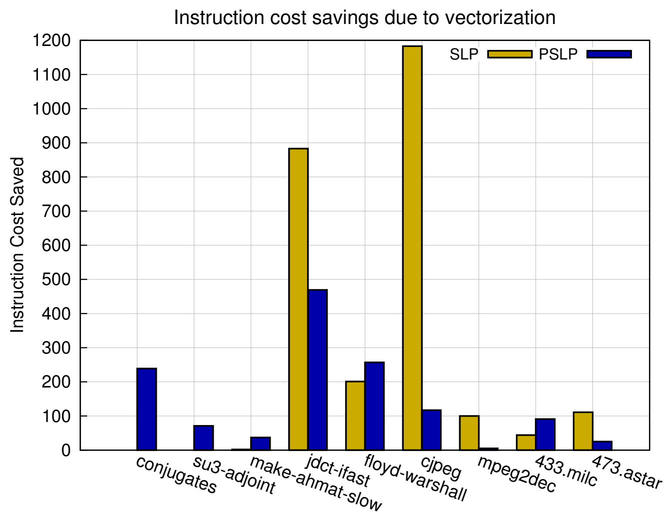


Figure 10. The total static code cost saved by each vectorizer over scalar, according to the cost model.

many times they generate code). SLP and PSLP complement each other. As shown in Figure 8, there are four possible cases: "No vectorization", "SLP", "PSLP extends SLP" and "PSLP only". The intersection "PSLP extends SLP" refers to the scenario where SLP would succeed even without PSLP, but it performs better with the help of PSLP (i.e., vectorization succeeds deeper in the SLP graph). We measure each of these cases (except for "No vectorization") across the benchmarks and we show the numbers in Figure 9.

These coverage results cannot be directly correlated to the performance results as they represent static data, not dynamic. The overall performance improvement achieved is a function of the frequency that the vectorized code gets executed (i.e., how "hot" it is) and the savings achieved compared with the original scalar code (i.e., the improvement estimated by the cost model). For example, Figure 9 shows that the kernels *conjugates* and *su3-adjoint* trigger PSLP a similar number of times, but *conjugates* takes 0.37× the baseline time, whereas *su3-adjoint* takes 0.97×. This is because the average cost savings for each vectorization success in *conjugates* is 3.4× that of *su3-adjoint* (Figure 10).

In addition, the coverage results are affected by the amount of unrolling performed. As mentioned in Section 4, the kernels are compiled with more aggressive unrolling which causes the techniques to succeed more times compared to the whole benchmarks. Nevertheless, the coverage results clearly show that PSLP improves the coverage of vectorization across the majority of codes shown.

5.3 Static Instruction Savings

The cost model of the vectorization algorithm (Section 3.4) makes sure that vectorization is only applied if it leads to code with lower cost. As already mentioned, the cost calculated by the cost model is very close to the actual count

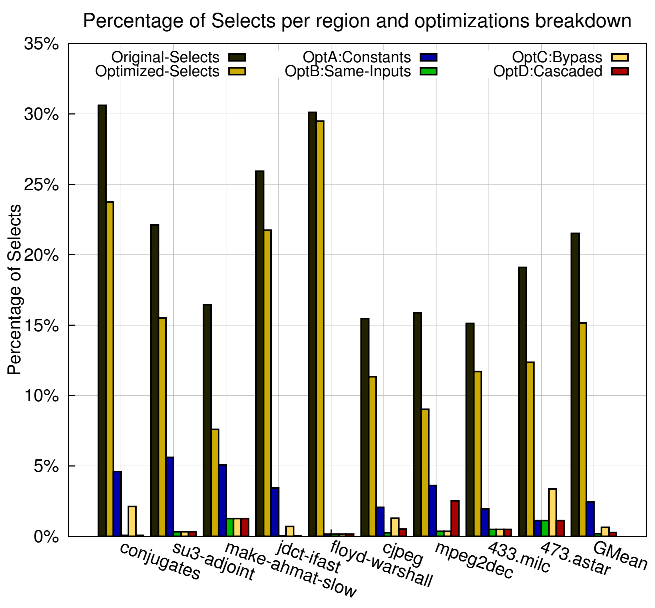


Figure 11. Number of select instructions emitted for the vectorized region, before and after select removal optimizations. Also the number of Selects removed by each optimization.

of instructions (it is usually weighted by the instruction latency). Therefore the vectorizer is only triggered if it leads to faster code with fewer instructions. The total cost saved by each of the vectorizers is shown in Figure 10. These savings are measured at the intermediate representation (IR) level. By the time the final code gets generated at the back-end the target code might look significantly different and the savings might differ.

5.4 Select Instructions

To get more insight on how PSLP performs, we measured how many select instructions are emitted in regions of code that get successfully vectorized by PSLP (Figure 11). To be more precise, the regions include all the instructions that form the PSLP graphs. We measured the number of selects originally emitted by PSLP just to maintain correctness ("Original-Selects"), the number of Selects that remain after all the select-removal optimizations described in Section 3.3.4 ("Optimized-Selects") and the number of Selects removed by each of the optimizations (OptA to OptD correspond to the descriptions in Figure 4).

According to Figure 11, the number of select instructions introduced by PSLP is significant (about 21% of the code in the PSLP graphs, on average). The select-removal optimizations manage to remove about 28% of the total selects originally emitted by PSLP, bringing them down to 15%. The most effective of these optimizations removes selects between constants (OptA).

5.5 Summary

We have evaluated PSLP on a number of kernels and full benchmarks, showing that it can reduce the execution-time down to 0.37×. It achieves this by increasing the amount of code that would have been vectorized by SLP anyway, as well as enabling SLP to vectorize code that it would have otherwise left as scalar.

6. Related Work

6.1 Vector Processing

Various commercial (for example [24, 29]) and experimental (e.g., [11]) wide vector machines have been built in the past. These machines were used to accelerate scientific vector code, usually written in some dialect of Fortran.

More recently short SIMD vectors have become a standard feature of all commodity processors for most desktop and mobile systems. All major processor manufacturers (Intel, AMD, IBM and ARM) support some sort of short-vector ISA (e.g., MMX/SSE*/AVX/AVX2 [10], 3DNow! [23], VMX/AltiVec [9] and NEON [3] respectively). These ISAs are constantly under improvement and get updated every few years with more capable vector instructions and/or wider vectors.

Modern graphics processors (GPUs), like old vector machines, implement hardware vectorization [15]. They do so by executing groups of 32 (on Nvidia) or 64 (on AMD) adjacent threads in *warps* in lock-step. Such large vector widths are possible thanks to data-parallel input languages like CUDA or OpenCL, where the programmer explicitly exposes the available parallelism to the hardware. This effectively overcomes intrinsic limitations of compiler-based analysis, leading to substantial runtime and energy improvements over traditional CPU execution for suitable workloads.

6.2 Loop Vectorization

Loops are the main target of vectorization techniques [32]. The basic implementation strip-mines the loop by the vector factor and widens each scalar instruction in the body to work on multiple data elements. The works of Allen and Kennedy on the Parallel Fortran Converter [1, 2] solve many of the fundamental problems of automatic vectorization. Numerous improvements to the basic algorithm have been proposed in the literature and implemented in production compilers. Efficient run-time alignment has been proposed by Eichenberger et al. [6], while efficient static alignment techniques were proposed by Wu et al. [33]. Ren et al. [27] propose a technique that reduces the count of data permutations by optimizing them in groups. Nuzman et al. [21] describe a technique to overcome non-contiguous memory accesses and a method to vectorize outer loops without requiring loop rotation in advance [20].

An evaluation of loop vectorization performed by Maleki et al. [17] shows the limits of current implementations. State-of-the-art compilers, like GCC and ICC, can vectorize only a small fraction of loops in standard benchmarks like *Media Bench*. The authors explain these poor results as (1) lack of accurate compiler analysis, (2) failure to perform preliminary transformations on the scalar code and (3) lack of effective cost models.

6.3 SLP Vectorization

Super-word level parallelism (SLP) has been recently introduced to take advance of SIMD ISAs for straight-line code. Larsen and Amarasinghe [13] were the first to present an automatic vectorization technique based on vectorizing parallel scalar instructions with no knowledge of any surrounding loop. Variants of this algorithm have been implemented in all major compilers including GCC and LLVM [28]. This is the state-of-the-art SLP algorithm and in this paper we use its LLVM implementation as a baseline for comparison and as a starting-point for our PSLP work.

Shin et al. [30] introduce an SLP algorithm with a control-flow extension that makes use of predicated execution to convert the control flow into data-flow, thus allowing it to become vectorized. They emit `select` instructions to perform the selection based on the control predicates.

Other straight-line code vectorization techniques which depart from the SLP algorithm have also been proposed in the literature. A back-end vectorizer in the instruction selection phase based on dynamic programming was introduced by Barik et al. [4]. This approach is different from most of the vectorizers as it is close to the code generation stage and can make more informed decisions on the costs involved with the instructions generated. An automatic vectorization approach that works on straight-line code is presented by Park et al. [25]. It succeeds in reducing the overheads associated with vectorization such as data shuffling and inserting/extracting elements from the vectors. Holewinsky et al. [8] propose a technique to detect and exploit more parallelism by dynamically analyzing data dependences at runtime, and thus guiding vectorization. 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 tree.

None of these approaches identify the problem of mismatching instructions while attempting to build vectors, nor do they try to solve it in any way. The PSLP approach of introducing selectively-executed padded code to solve this mismatching problem is unique, to the best of our knowledge.

6.4 Vectorization Portability

Another relevant issue for vectorization is portability across platforms. The various types of SIMD instructions available on different architectures require the definition of suitable abstractions in the compiler's intermediate representation. These must be general enough to embrace various vectorization patterns without sacrificing the possibility of generating efficient code. Nuzman et al. targeted this problem by proposing improvements to the GIMPLE GCC intermediate representation [19] and through JIT compilation [22].

7. Conclusion

In this paper we presented PSLP, a novel automatic vectorization algorithm that improves upon the state-of-the-art. PSLP solves a major problem of existing SLP vectorization algorithms, that of purely relying on the existence of isomorphic instructions in the code. Our technique transforms the non-isomorphic input code graphs into equivalent isomorphic ones in a near-optimal way. This is performed by careful *instruction padding* while making sure that the program semantics remain unchanged. The end result is padded code which gets successfully vectorized when the state-of-the-art techniques would either fail or partially vectorize it. The evaluation of our technique on an industrial-strength compiler and on a real machine shows improved coverage and performance gains across a range of kernels and benchmarks.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) through grant reference EP/K026399/1.

References

[1] J. R. Allen and K. Kennedy. *PFC: A program to convert Fortran to parallel form.* Rice University, Department of Mathematical Sciences, 1981.

[2] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *Tranactions on Programming Languages and Systems (TOPLAS)*, 9(4), 1987.

[3] ARM Ltd. ARM NEON. http://www.arm.com/products/processors/technologies/neon.php, 2014.

[4] R. Barik, J. Zhao, and V. Sarkar. Efficient selection of vector instructions using dynamic programming. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2010.

[5] H. Bunke, X. Jiang, and A. Kandel. On the minimum common supergraph of two graphs. *Computing*, 65(1), 2000.

[6] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[7] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. Mediabench II video: Expediting the next generation of video systems research. *Microprocessors and Microsystems*, 2009.

[8] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[9] IBM PowerPC Microprocessor Family. Vector/SIMD Multimedia Extension Technology Programming Environments Manual, 2005.

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

[11] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *Computer*, 30 (9), 1997.

[12] V. S. Lapinskii, M. F. Jacome, and G. A. De Veciana. Cluster assignment for high-performance embedded VLIW processors. *Transactions on Design Automation of Electronic Systems (TODAES)*, 2002.

[13] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2000.

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

[15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. 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. A compiler framework for extracting superword level parallelism. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[17] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[18] J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1), 1982.

[19] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

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

[21] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[22] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2011.

[23] S. Oberman, G. Favor, and F. Weber. AMD 3DNow! technology: Architecture and implementations. *Micro, IEEE*, 19(2), 1999.

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

[25] Y. Park, S. Seo, H. Park, H. Cho, and S. Mahlke. 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)*, 2012.

[26] L.-N. Pouchet. PolyBench: The polyhedral benchmark suite. http://www.cs.ucla.edu/˜pouchet/software/polybench/, 2012.

[27] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2006.

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

[29] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1), 1978.

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

[31] SPEC. Standard Performance Evaluation Corp Benchmarks. http://www.spec.org, 2014.

[32] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

[33] P. Wu, A. Eichenberger, and A. Wang. Efficient SIMD code generation for runtime alignment and length conversion. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2005.