The Journal of Supercomputing (2020) 76:4731–4746
https://doi.org/10.1007/s11227-019-03057-4

# Vectorizing programs with IF-statements for processors with SIMD extensions

**Huihui Sun**[1] · **Sergei Gorlatch**[1] · **Rongcai Zhao**[2]

Published online: 11 November 2019


**Abstract**
Vectorization of programs is crucial for achieving high performance on modern processors with SIMD (Single Instruction Multiple Data) extensions. Programs with IF-statements suffer from control flow divergence that seriously complicates automatic vectorization. Therefore, contemporary compilers employ the IF-conversion approach to convert control flow to data flow, which relies on using predicated execution techniques (i.e., masked or select SIMD instructions). In this paper, we enhance the compiler's capabilities to generate efficiently vectorized code for processors without masked instructions. We improve the state of the art in program vectorization by developing a novel approach—IF-select transformation—which is applicable to arbitrarily nested IF-statements. We implement our approach in the open-source Open64 compiler and evaluate its performance on the SW26010 processor used in the Sunway TaihuLight supercomputer (currently #3 in the TOP500 list) that does not support masked instructions. We extend our vectorization approach by providing an additional LLVM optimization pass to reduce the amount of masked memory accesses on processors without masked instructions, e.g., IBM Power8 and ARMCortex-A8. Experimental results demonstrate the performance advantages of the suggested vectorization techniques.

**Keywords** Compiler optimization · SIMD extensions · IF-conversion · Masked instructions · Select instructions

✉ Huihui Sun
huihuisun@uni-muenster.de

[1] University of Münster, Münster, Germany

[2] National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China

## 1 Introduction

Most modern processors are equipped with *single-instruction multiple data* (*SIMD*) extensions that can operate on vectors of values in parallel to enable high performance. To exploit this performance potential, programs must be transformed to a form with SIMD instructions; this is traditionally called *vectorization*. Manual vectorization by the programmer using hand-written intrinsics is tedious, error-prone and non-portable. Therefore, automatic vectorization is an indispensable component in most modern compilers, such as the open-source compilers Open64 [2], GCC [8] and LLVM [15] and the commercial compiler ICC [12].

There are two classic approaches to vectorization: (1) loop vectorization [23] combines multiple occurrences of a scalar operation across consecutive loop iterations into one SIMD instruction, (2) basic block or superword-level parallelism (SLP) vectorization [14] transforms a group of isomorphic operations into one SIMD instruction. Most of the mainstream compilers, such as GCC and LLVM, implement both approaches. Another increasingly popular approach SPMD-on-SIMD [19] —which implements single program multiple data (SPMD) execution on the SIMD extensions—converts multiple instances of a kernel into one SIMD instruction. Compilers implementing SPMD-on-SIMD include the Intel ISPC compiler [19], whole-function vectorizer (WFV) [13] and region vectorizer (RV) [17].

IF-statements in a program introduce control flow divergence, which hampers vectorization. A typical solution to overcome this problem in the process of vectorization is IF-conversion [1] that converts control flow into data flow. This requires support from the targeted processors for predicated execution: both branches of an IF-statement are executed one after the other, with conditional execution of instructions based on the value of a Boolean expression, referred to as predicate or mask. Most SIMD instruction sets support predicated execution by providing masked instructions and/or select instructions. Masked instructions enable each branch of an IF-statement to execute one after the other, conditionally operating on the elements depending on the corresponding mask bits. While when using select (also called "blending", but we use "select" in the paper) instructions, both branches are executed in an unmasked manner, and select instructions use a mask to blend the variables from both branches before storing to memory. According to the Intel manual [11], select is faster than masked instruction, but select cannot be used when the operations in the IF-statements have exceptions, in other words, it is not always safe to use select instructions. This is because when using select, both branches are executed in an unmasked manner; thus, it does not mask exceptions, which may occur on the unmasked data. SIMD instruction sets supporting both masked and select instructions include Intel AVX512, ARM SVE, and RISC-V, while some SIMD instruction sets, e.g., ARM NEON and IBM VSX, only support select instructions. It is the compilers's responsibility to generate masked or select instructions, but this process involves intricate analysis and transformations.

For processors without masked instructions, current compilers often fail to fully utilize the targeted instruction sets for vectorizing programs with

IF-statements. In this paper, we target SW26010 processors with dedicated SIMD instruction sets using the open-source Open64 compiler. When vectorizing programs with IF-statements, the Open64 compiler is limited to single-level ones, thus, leaving many IF-statements in real applications unvectorized. To alleviate this problem. we develop a novel IF-conversion approach—*IF-select transformation*—that generates efficiently vectorized code targeting SW26010 processors for loops with arbitrarily nested IF-statements. Furthermore, we extend our approach within the LLVM compiler, as follows. When vectorizing programs with IF-statements, the LLVM compiler proceeds in two phases. First, during the target-independent intermediate represnetation (IR) phase, LLVM generates select IR instructions when the IF-statements prove (via analysis) to have no exceptions, otherwise, it generates masked IR instructions. Second, during the backend code generation phase, LLVM generates masked instructions for masked IR if the target supports it, otherwise (for the targets without masked instructions) it generates scalar code guarded by IF-cascades, which restricts the performance of vectorization. We extend our vectorization approach by introducing an LLVM optimization pass that reduces the amount of masked memory accesses on processors without masked instructions by using select instructions.

Specifically, we make the following contributions.

- We develop a novel IF-select transformation method which vectorizes loops with arbitrarily nested IF-statements in the Open64 compiler targeted for SW26010 processors [9] as used in the Sunway TaihuLight supercomputer (currently #3 in the TOP500 list [28]). We verify the efficiency of our approach by its experimental evaluation on a set of benchmarks from SPEC CPU2006 [10] containing loops with IF-statements.
- We introduce an optimized code generation technique that utilizes select instructions for reducing masked memory accesses targeting processors without masked instructions. We demonstrate a significant performance improvement compared to the exisitng techniques on IBM Power processors.

In the remainder of the paper, Sect. 2 provides background information on vectorization and on our target SIMD extensions. Section 3 presents our approach to vectorizing IFs using the Open64 compiler for SW26010 processors, and Sect. 4 presents our LLVM code generation optimization for IFs targeting IBM Power processors. Experimental results are presented in Sect. 5. Section 6 compares our approach to related work, and Sect. 7 concludes the paper.

## 2 Motivation

### 2.1 SIMD extensions and vectorization

In this paper, we target modern processors with SIMD extensions. We use the SW26010 processor as our sample target architecture: each core of it employs a 256-bit SIMD extension that works on 256 bits in parallel: it can be one long int (256-bit)

operation, or 8 integer operations, or 4 double-precision floating point operations. Without loss of generality, we assume in this paper that 4 operations are executed simultaneously on 64-bit floating point values.

Figure 1 illustrates a simple vectorization example. On the left-hand side, Fig. 1a shows a loop which is straightforwardly vectorizable. On the right-hand side, Fig. 1b shows the result of vectorization using SIMD intrinsics, i.e., C-style functions providing access to SIMD instructions. For simplicity, in this paper, we will call these intrinsics *instructions*. A SIMD extension executes multiple loop iterations in Fig. 1b in parallel as follows: load the operands from memory to vectors, add the two vectors and store the result vector back into memory.

Table 1 shows the SIMD instructions used in this paper, with the names as used in the SW26010 processor. Other modern CPUs with SIMD extensions have very similar instructions. In the table, we only list the instructions for double-precision floating point parameters; here, the vector type `doublev4` means 4 packed 64-bit double elements.

### 2.2 Vectorizing IF-statement: select instructions versus masked instructions

The existing approaches to vectorizing IF-statements rely on predicated execution, which most SIMD instruction sets support via select instructions and/or masked instructions. Some processors, like Intel Xeon processors, support both select and masked instructions, while some processors, like IBM Power processors, only support select instructions. In this subsection, we take Intel instructions to illustrate the differences between select and masked instructions when used to vectorize IF-statements.

Figure 2a shows a loop with an IF-statement. Figure 2b shows the vectorized loop using select instructions. When vectorizing IFs with select instructions, operations are executed in an unmasked manner and mask is only used to blend the variables contain the correct values for both branches, e.g., _mm256_blendv_pd in the figure; exceptions cannot be masked. Figure 2c shows the vectorized loop using masked instructions: every memory access instruction is associated with a mask that controls which element shall be modified by the instruction; thus, exception can be masked. Given a targeted processor without masked instruction, if the compiler is not certain

```
for(i=0;i<1024;i++)
{
  c[i]=a[i]+b[i];
}
```

**(a)**

```
for(i=0;i<1024;i=i+4)
{
  simd_load(v_a,&a[i]);
  simd_load(v_b,&b[i]);
  v_c=simd_vaddd(v_a,v_b);
  simd_store(v_c,&c[i]);
}
```

**(b)**

**Fig. 1** **a** An easily vectorizable loop, **b** the loop after vectorization

**Table 1** Specific SIMD instructions used in this paper

| Instruction | Operation | Input | Output | Functional description |
|---|---|---|---|---|
| simd_load | Load | doublev4 va, double *addr | void | Load 4 double elements into vector va from contiguous memory starting from *addr |
| simd_store | Store | doublev4 va, double *addr | void | Store 4 elements of vector va into contiguous memory starting from *addr |
| simd_vaddd | Addition | doublev4 va, vb | doublev4 | Add 4 elements of va with 4 elements of vb element-wise, return the result |
| simd_vsubd | Subtraction | doublev4 va, vb | doublev4 | Subtract 4 elements of va from 4 elements of vb element-wise, return the result |
| simd_vseleq | Select | doublev4 va, vb, vc | doublev4 | Test the value of va element-wise: if it equals 0, then return the element of vb, otherwise return the element of vc |
| simd_vfcmplt | Comparison | doublev4 va, vb | doublev4 | Compare the value of va and vb element-wise; if va < vb, then the element of vc is assigned 1.0, otherwise 0 |

about the absence of exceptions, it emits scalar code guarded by IF-cascades, i.e., load or store operation is guarded by a compare on the particular lane's mask value and a branch instruction, so that the memory operation is only performed if the lane's value in the mask is true.

### 2.3 Our approach: the idea

For a given target SIMD instruction set, the existing compilers decide which instructions to generate for vectorizing IF-statements. As select instructions are faster in general than masked instructions, modern compilers tend to employ select instructions when the program analysis confirms that there are no exceptions. If there might be exceptions, the compilers generate masked instructions if the target permits them, otherwise, the compilers generate scalar code guarded by IF-cascade for targets without masked instructions. There are two main reasons why the compilers do not generate select instructions, which results in IF-cascades for targets without masked instructions. First, the program analysis for proving the absence of exception is too pessimistic and may provide false negative results. Second, the capability of transforming IF-statements into select instructions is limited for complex nested IF-statements.

In this paper, we extend the scope of generating select instructions for vectorizing IF-statements when targeting processors without masked instructions. We improve the transformation capability of the Open64 compiler by presenting a new IF-conversion approach—IF-select transformation based on statement matching on the IR level—which works for arbitrarily nested IFs. In addition, we introduce a code generation optimization technique for the LLVM compiler to reduce masked memory access instructions in LLVM-IR to select instructions with normal

```
for(i=0;i<1024;i++){
  if(m[i]<0)
    c[i]=a[i]+b[i];
  else
    c[i]=0;
}
```

**(a)**

```
for(i=0;i<1024;i=i+4){
  v_0=_mm256_set1_pd(0);
  v_m=_mm256_load_pd(m+i*4);
  mask=_mm256_cmp_pd(v_m,v_0,1);
  v_a=_mm256_load_pd(a+i*4);
  v_b=_mm256_load_pd(b+i*4);
  v_add=_mm256_add_pd(v_a,v_b);
  v_c=_mm256_blendv_pd(v_add,v_0,mask);
  _mm256_store_pd(c+4*i,v_c);
}
```

**(b)**

```
for(i=0;i<1024;i=i+4){
  v_0=_mm256_set1_pd(0);
  v_m=_mm256_load_pd(m+i*4);
  mask=_mm256_cmp_pd(v_m,v_0,1);
  mask_neg=_mm256_cmp_pd(v_m,v_0,13)
  v_a=_mm256_maskload_pd(a+i*4,mask);
  v_b=_mm256_maskload_pd(b+i*4,mask);
  v_add=__mm256_add_pd(v_a,v_b);
  _mm256_maskstore_pd(c+i*4,mask,v_add);
  _mm256_maskstore_pd(c+i*4,mask_neg,v_0);
}
```

**(c)**

**Fig. 2** **a** A loop with an IF-statement, **b** vectorized loop using select instructions, **c** vectorized loop using masked instructions

(unmasked) memory access instructions when the target processor does not support masked instructions. The following two sections describe our approach in detail.

## 3 Vectorizing IF-statements using the Open64 compiler

The idea of our approach is that we generate select statements by matching the statements in the THEN block with the statements in the ELSE block, and we combine each pair of matched statements into a select statement. We say that statements are matched if they define the same variable. For example, in the statement `if(cond){dst=val1;} else{dst=val2;}`, the statements in the THEN and ELSE blocks both define `dst`, so they are matched, and we can combine them into one select statement `dst=select(cond,val1,val2)`. In contrast, if there are no matched statements in the THEN or the ELSE block, then we assume that there is a fictitious statement `dst=dst` to match with the current statement, and we combine the original statement with the fictitious statement into one select statement. For example, in the statement `if(cond){dst=val1;}`, there is no ELSE part and thus no matched statement; therefore, for this single IF-statement we generate the select statement `dst=select(cond,val1,dst)`. We denote the former case that generates a select statement for two matched IF-statements as *Rule 1*, and the latter case that generates a select statement for a single IF-statement is denoted as *Rule 2*.

Algorithm 1 shows the pseudocode of our *IF-select transformation* method applied to an IF-statement in a loop. We first create a new block `sel_wn` to store the newly generated select statements (line 2). Then, we sequentially traverse the statements in the THEN and ELSE blocks (line 6): we initialize the flag `matched` as FALSE (line 7) at the beginning of each traversal pass, and then we try to match the statements in the THEN and ELSE blocks and generate corresponding select statements according to Rule 1 and Rule 2 (line 8–44). Eventually, if `sel_wn` is not empty (line 45), we replace the original IF-statement with `sel_wn` (line 46); otherwise, we leave the IF-statement unchanged.

**Algorithm 1:** IF-select Transformation

```
1  Function IF-selectTransformation(IF)
2      build a new block sel_wn ;                                   // store the generated select statements
3      get the Array_Dependence_Graph as ADG;
4      then_stmt=get_first(IF.then);                                // initiate the current statement in the THEN block
5      else_stmt=get_first(IF.else);                                // initiate the current statement in the ELSE block
6      while then_stmt!=NULL | else_stmt!=NULL do
7          BOOL matched = FALSE;
8          if then_stmt != NULL then
9              else_iter=else_stmt;
10             while else_iter!=NULL & matched ==FALSE do
11                 if else_iter is matched with then_stmt then
12                     matched = TRUE ;                             // find the matched else_iter
13                 else
14                     else_iter=get_next(else_iter);
15             if matched==FALSE then                               // Case 1
16                 generate select statement (Rule 2) and insert it into sel_wn;
17                 then_stmt=get_next(then_stmt);
18             else
19                 if else_iter == else_stmt then                   // Case 2
20                     generate select statements (Rule 1) and insert it into sel_wn;
21                     then_stmt=get_next(then_stmt);
22                     else_stmt=get_next(else_stmt);
23                 else                                             // Case 3
24                     matched = FALSE;
25                     then_iter=then_stmt;
26                     while then_iter!=NULL & matched ==FALSE do
27                         if then_iter is matched with else_stmt then
28                             matched = TRUE ;                     // find the matched then_iter
29                         else
30                             then_iter=get_next(then_iter);
31                     if matched==FALSE then
32                         generate select statement (Rule 2) and insert it into sel_wn;
33                         else_stmt=get_next(else_stmt);
34                     else
35                         if Forward_Motion(then_stmt, then_iter, ADG) | Forward_Motion(else_stmt,
                             else_iter, ADG) then
36                             generate select statements (Rule 1) and insert it into sel_wn;
37                             then_stmt=get_next(then_stmt);
38                             else_stmt=get_next(else_stmt);
39                         else
40                             sel_wn = NULL;
41                             return;
42         else if else_stmt != NULL then                           // Case 4
43             generate select statement (Rule 2) and insert it into sel_wn;
44             else_stmt=get_next(else_stmt);
45     if sel_wn != NULL then
46         replace IF with sel_wn;
```

We describe in the following how we match statements and generate select statements, especially when there are flow dependences in the block. We begin with traversing the ELSE block from the current statement and looking for a matching statement (line 10–14) in the THEN block. If there is no matching statement (Case 1), then we generate a select statement according to Rule 2 (line 16), and we turn to the next statement in the THEN block (line 17). If we find a matching statement that is the current statement in the ELSE block (Case 2), then we combine these two statements and generate a select statement according to Rule 1 (line 20), and then we turn to the next statements in the THEN and ELSE blocks (line 21–22).

In the case when the matching statement is not the current statement in the ELSE block (Case 3), we reset flag `matched` to FALSE (line 24), and we turn to looking for a matching statement in the THEN block (line 26–30) for the current statement in the ELSE block. If there is no matching statement in the THEN block (line 31), then we generate a select statement according to Rule 2 (line 32), and we turn to the next statement in the ELSE block (line 33). If a matching statement for the current statement is found, then it means that the order of these two statements is different in the THEN and ELSE blocks: e.g., `dst1` is defined before `dst2` in the THEN block and after `dst2` in the ELSE block. In this case, we check whether there is a flow dependence between the memory accesses in these two statements (line 35). If no flow dependence is found from `then_stmt` to `then_iter`, then we change the order of these two statements in the THEN block by moving `then_stmt` after `then_iter`, likewise for ELSE block. Otherwise, we retain the IF-statement unchanged, ignore all select statements generated before, and return (line 41). After detecting flow dependences and reordering statements, we generate select statements according to Rule 1 (line 36) and turn to the next statements in the THEN and ELSE blocks (line 37–38). Note that case 3 enables us to generate select statements even when there is a flow dependence either in the THEN or in the ELSE block. If we would simply add all matched statements to `sel_wn` and perform an analysis for detecting a cyclic dependence afterward, we may end up with inconsistent semantics because of ignoring flow dependences.

If we are done with all statements in the THEN block and there are still statements in the ELSE block (Case 4), then for every statement in the ELSE block we generate a select statement according to Rule 2 (line 43).

To illustrate our approach, we take as an example the loop with an IF-statement in Fig. 3 and we denote the condition `a[i]<b[i]` by `cond` in the following. We start by looking for a matched statement in the ELSE block for the current statement in the THEN block (line 5). Because there is no matching statement as in Case 1 of Algorithm 1, we generate statement h(i) = select(cond,k(i),h(i)). For the next

```
for(i=0;i<1024;i++)
{
  if(a[i]<b[i])
  {
    h(i)=k(i);
    b(i)=c(i);
    l(i)=m(i);
    u(i)=v(i);
  }
  else
 {
    b(i)=d(i);
    f(i)=g(i)
    u(i)=w(i);
    l(i)=n(i);
    x(i)=y(i);
  }
}
```

**(a)**

```
for(i=0;i<1024;i++)
{
  h(i)=select(a[i]<b[i],k(i),h(i));
  b(i)=select(a[i]<b[i],c(i),d(i));
  f(i)=select(a[i]<b[i],f(i),g(i));
  u(i)=select(a[i]<b[i],v(i),w(i));
  l(i)=select(a[i]<b[i],m(i),n(i));
  x(i)=select(a[i]<b[i],x(i),y(i));
}
```

**(b)**

**Fig. 3** **a** A loop with a loop-dependent IF-statement, **b** the same loop after IF-select transformation

statement (line 6), we find that the current statement in the ELSE block (line 12) matches, which corresponds to Case 2, so we generate b(i) = select(cond, c(i), d(i)). For the next statement in the THEN block (line 7), the matching statement (line 15) is not the current statement in the ELSE block, so we turn to deal with the statements in the ELSE block as Case 3. For the current statement in the ELSE block (line 13), we cannot find a matching statement in the THEN block, so we generate f(i) = select(cond,f(i),g(i)). For the next statement (line 14), we find a matching statement (line 8), and since there are no dependences between the memory accesses of these two statements, we generate u(i) = select(cond,v(i),w(i)) and l(i) = select(cond,m(i),n(i)). Finally, we generate x(i) = select(cond, x(i),y(i)).

We further extend our IF-select transformation method (Algorithm 1) to handle nested loop-dependent IF-statements: we tackle the IF-statements starting from the innermost one and moving to the outermost.

Figure 4a illustrates how we vectorize a nested loop-dependent IF-statement. According to Algorithm 1, we first transform the innermost IF-statement to a select statement (Rule 1), with the result in Fig. 4b. Then, we transform the outermost IF-statement to a select statement (Rule 2), with the result in Fig. 4c. Finally, we generate SIMD instructions as shown in Fig. 4d.

## 4 The optimized LLVM code generation for vectorizing IFs

When processing IF-statements in LLVM, the compiler analyzes in a pessimistic way whether there might be exceptions. When it is proved that there are no exceptions, the compiler hoists load operations before the condition of the IF-statement

```
for(i=0;i<1024;i++)
{
 if(a[i]<b[i])
 {
  if(a[i]<10)
    c[i]=a[i]+b[i];
  else
    c[i]=a[i]-b[i];
  }
}
```

**(a)**

```
for(i=0;i<1024;i++)
{
 if(a[i]<b[i])
 {
   c[i]=select(a[i]<10,a[i]+b[i],
        ↪ a[i]-b[i]);
  }
}
```

**(b)**

```
for(i=0;i<1024;i++)
{
 c[i]=select(a[i]<b[i],select(a[i]<10,a[i]+
      ↪ b[i],a[i]-b[i]),c[i]);
}
```

**(c)**

```
for(i=0;i<1024;i=i+4)
{
 simd_load(v_a,&a[i]);
 simd_load(v_b,&b[i]);
 v_add=simd_vaddd(v_a,v_b);
 v_sub=simd_vsubd(v_a,v_b);
 v_cond1=simd_vfcmplt(v_a,v_10);
 v1=simd_vseleq(v_cond1,v_sub,v_add);
 simd_load(v_c,&c[i]);
 v_cond2=simd_vfcmplt(v_a,v_b);
 v2=simd_vseleq(v_cond2,v_c,v1);
 simd_store(v2,&c[i]);
}
```

**(d)**

**Fig. 4** **a** A loop with a nested loop-dependent IF-statement, **b** apply IF-select transformation to the innermost IF-statement, **c** apply IF-select transformation to the outermost IF-statement, **d** vectorized code

is evaluated, and it sinks store operations afterthe IF-statement. With load and store operations hoisted/sunk out of the IF-statements, the memory access operations are no long predicated, and select instructions are emitted to blend the values from both branches of the IF-statement. Otherwise, the memory access operations are predicated, and masked instructions are generated for them in the IR level.

At the backend, the Scalarize Masked Mem Intrin (SMMI) pass of LLVM queries the Target Transform Information (TTI) analysis results to evaluate if masked instructions are legal for the selected target. In case of a negative answer, the pass transforms the masked instructions to scalar loads and stores in IF-cascade guarded scalar code. Therefore, this approach decreases the overall performance significantly.

To address this problem, we introduce an additional code generation optimization pass to utilize select instructions for reducing masked memory access LLVM-IR instructions for targets without masked instructions. In order to prevent exceptions, we avoid out-of-bound memory access using padding. Each memory allocation is padded to the next multiple of the hardware's vectorization factor, for example, on a processor with a 256-bit SIMD extension, an allocation is padded up to the next 32 bytes. This padding has negligible effect on the efficiency of memory allocation. After we prevent out-of-bound memory access, we can optimize masked memory access IR to select instructions at the backend.

Our optimization performs the following steps. (1) Create an unmasked vectorized load based on the masked load. (2) Cast the original mask to integer data type and compare it with 0. Guard the vectorized load with a branch to ensure that the mask is not equal to 0. When the mask is 0, the access may result in an access violation. (3) Create a select instruction that selects between the loaded values and the original values based on the original mask. (4) Remove the masked load instructions. For masked store instructions, the optimization performs the similar following steps. (1) Create a vectorized load at the memory region where the store will be performed. (2) Create a select instruction to select between the original values intended to be stored and the previously loaded values according to the mask. (3) Replace the masked store instruction with an unmasked vectorized store of the selected values. (4) Just as with masked load, secure the operation with a check if the mask is not equal to 0. The optimization results in a higher performance than the IF-cascades generated by the existing SMMI pass.

## 5 Experimental evaluation

### 5.1 Using the Open64 compiler on SW26010

To evaluate our approach of the Open64 compiler targeting SW26010 processors, we conduct our experiments on the programs with IF-statements from the SPEC CPU2006 benchmark suite [10], listed in Table 2. Out of 29 programs in SPEC CPU2006, the six programs in the table contain IF-statements in their most

**Table 2** Benchmark kernels with IF-statements from SPEC CPU2006

| Program | Kernel | Kernel runtime (%) | Application category | IF-stmt type |
|---|---|---|---|---|
| 429.mcf | primal_bea_mpp | 49.95 | Combinatorial optimization | Nested |
| 456.hmmer | P7Viterbi | 99.53 | Search gene sequence database | Nested |
| 464.h264ref | SetupFastFullPelSearch | 40.93 | Video compression | Nested |
| 482.sphinx3 | vector_gautbl_eval_logs3 | 38.67 | Speech recognition | Single |
| 458.sjeng | std_eval | 15.11 | Pattern recognition | Nested |
| 462.libquantum | quantum_toffoli | 63.41 | Physics and quantum computing | Nested |

time-consuming loops. Our experimental platform is a SW26010 processor with a 256-bit dedicated SIMD extension, running under Linux Redhat Enterprise 5.

For the six benchmarks in Table 2, we measure both kernel and whole-program speedups. We compare our approach against the original Open64 compiler vectorization. All programs are compiled with the same flags: -O3, -LNO:simd=1. *-O3* enables global optimization, code generator and loop nest optimization. *-LNO:simd=1* enables loop vectorization. The execution time of a kernel or program is calculated as the average of 20 runs (the measured results are within few percent over each run). The speedups are calculated as compared with the execution on the same SW26010 processor, but without vectorization.

Figure 5 shows the kernel speedups. The average kernel speedup achieved by our approach is 1.45 × compared to the non-vectorization baseline and 1.26 ×

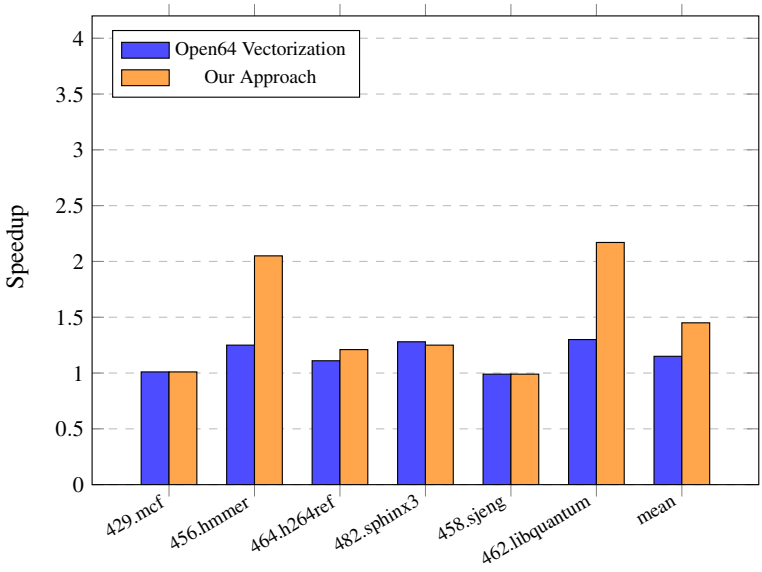


**Fig. 5** Kernel speedups: our approach compared with the Open64 vectorization

compared to the Open64 vectorization. Note that the kernels we evaluate here are notoriously difficult to vectorize because of control flow or irregular memory access or both. Our approach outperforms Open64 vectorization for three out of six benchmark programs and matches it for the three remaining programs. We explain the achieved performance gains as follows. For 456.hmmer, our IF-select transformation is applied to the innermost loop-dependent IF-statement, the same is done for 464.h264ref. For 462.libquantum, our IF-select transformation is applied to the two-level nested IF-statement. Our approach achieves a speedup similar to the Open64 vectorizer for 482.sphinx3, because its IF-statement is not nested. The remaining two programs which show no improvement are 429.mcf and 458.sjeng: they are not vectorized. In 429.mcf, its IF-statement contains pointers where dependence cycles are conservatively assumed and, therefore, the surrounding loop is excluded from vectorization. In 458.sjeng, there is a three-level nested, loop-dependent IF-statement; however, the dependence cycles between the indirected arrays exclude the loop from vectorization.

The reason why achieved speedup falls short of the high performance potential offered by the hardware of SIMD extensions can be explained by two aspects. First, the IF-select transformation introduces overhead by executing both branches of the vectorized IF-statement. Second, if the memory access patterns in a kernel are irregular, then when vectorizing the loop, it requires additional cost to pack the memory access into vectors compared with simply load the memory into vectors in case of a continuous memory access.

Figure 6 shows the whole-program speedups. The average whole-program speedup achieved by our approach is 1.28 × as compared to the non-vectorization baseline and 1.12 × as compared to the Open64 vectorization. In most cases, the

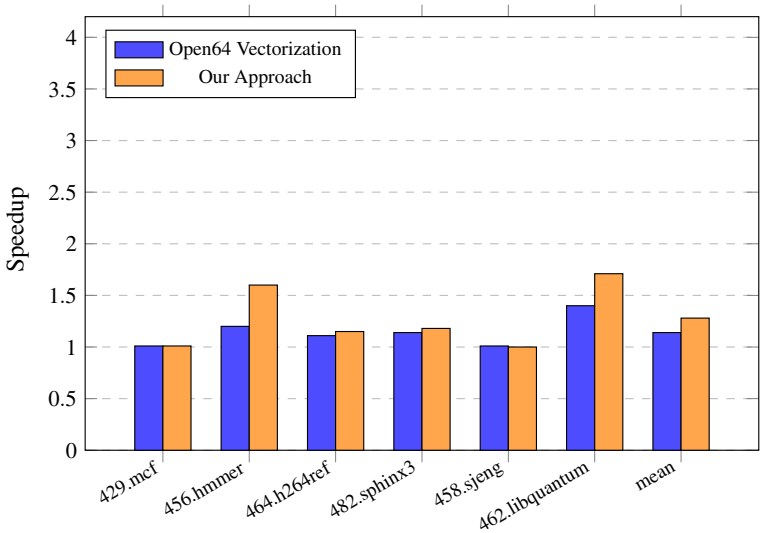


**Fig. 6** Whole-program speedups: our approach vs. Open64 vectorization

achieved whole-program speedups are consistent with the cumulative speedups of the most time-consuming kernels.

### 5.2 Using LLVM on the IBM power processor

To evaluate our presented approach of lowering masked memory access LLVM-IR to select instructions, we compare the kernel runtimes of several benchmarks taken from the SHOC [7] (triad, reduction and md5hash) and the Rodinia [5] (cfd, hotspot and lavaMD) benchmark suites. Each kernel is run 30 times to reduce the noise from caches, etc. We perform the experiment on a Power8 processor, equipped with IBM AltiVec vector instructions. We present average runtime results for all benchmarks. The baseline for all benchmarks is the unvectorized kernel optimized only with the standard LLVM optimization pipeline (-O3).

Figure 7 shows the results of LLVM with and without our optimization. Our approach achieves an average speedup of 1.66x compared to non-vectorized version, and a speedup of 1.23x compared to the SMMI results where our approach is disabled. The results verify the efficacy of our approach which generates select instructions instead of IF-cascades on the target IBM Power processor.

## 6 Related work

IF-statements are ubiquitous in high-performance computing applications, and their existence has seriously hindered the efficiency of vectorization. If control flow can be eliminated or converted to a simple data dependency, the vectorization efficiency is greatly increased, and the compiler's optimization performance is improved.

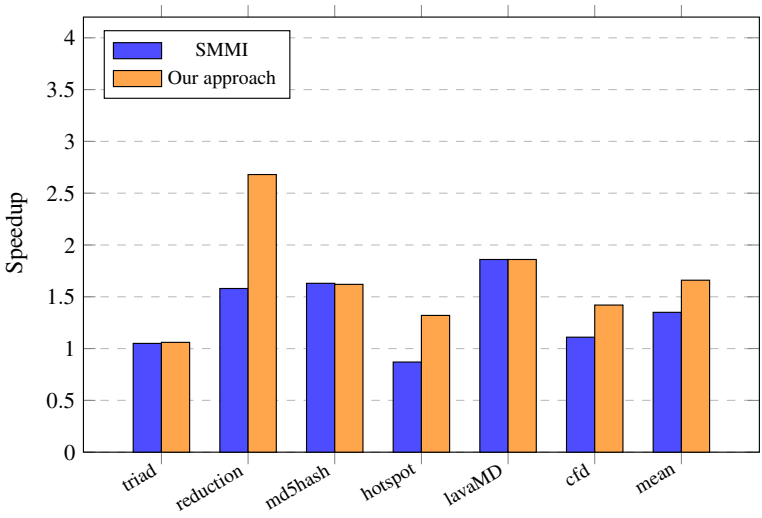


**Fig. 7** Speedups on IBM Power8

Cocke and Allen [27] proposed the technique of Loop Unswitching, which reduces or eliminates the control flow in a loop by hoisting control flow with a loop-independent conditional outside of the loop. The advantage of this technique is that it achieves efficient scheduling, convenient register allocation and good execution speed. At the same time, it can reduce the number of branches and increase the chance of exploiting loop vectorization [6]. However, Loop Unswitching may imply code bloat, which hinders the compiler to do other optimizations. Lokuciejewski et al. [16] proposed a method of Loop Unswitching optimization based on worst-case execution time (WCET), which can effectively alleviate the code bloat problem. Barton et al. [3] used an index set splitting technique to divide a loop containing a loop-dependent condition into several equivalent cycles without control flow. Our method is orthogonal to these methods.

Allen [1] proposed the IF-conversion technique. Its main idea is to convert control dependencies into data dependencies in order to unify all dependencies into one form. Bik et al. [4] used bit masking techniques to combine the values generated from different control flow branches. Shin et al. [21] proposed a method for IF vectorization based on SLP, followed by nested BOSCCs and special instructions to generate vectorized code for nested control flow [22]. Our approach proceeds differently from [21], where the IF-conversion [1] is applied to transform a program with IF-statements into an equivalent program with predicated statements, which are then transformed into select statements. This transformation relies on the predicate hierarchy graph (PHG) representing the nesting relations among predicates. Our approach generates select statements directly, without generating predicated statements; we also avoid building and analyzing the PHG. Tanaka et al. [26] proposed a SIMD code generation technique based on maintaining the basic inter-block data dependency. This vectorization of the basic block is achieved without changing the structure of the control flow and does not eliminate the conditional branch.

Karrenberg and Hack [13] presented whole-function vectorization on intermediate code given by a control flow graph in static single assignment form, which is based on the conversion of control flow to data flow. Moll and Hack [18] presented partial linearization which only linearizes varying branches (different values for different threads) while preserving uniform branches (same value for different threads) and implemented in region vectorizer [17]. In addition, region vectorizer adds a so-called BOSCC gadget that can skip the execution of a masked branch if its mask is all-false before partial linearization. In our recent work [25], we present warp-coherent condition vectorization based on partial control flow linearization to mitigate the performance degradation of control flow divergence. Sujon et al. [24] generated SIMD instructions for statements containing control flow branches based on the path optimization strategy, but this method only supports vectorization of a single path for a given loop. Pohl et al. [20] proposed a set of solutions to generate efficient vector instructions in the presence of control flow for ARM NEON. While in this work, we focus on another two targets—IBM Power and SW26010 processors—and we take exceptions operations into consideration.

## 7 Conclusions and discussion

In this paper, we present two techniques to extend the compilers' capability to generate highly efficient vectorized code by utilizing select instructions in absence of masked instructions, rather than conservatively employing IF-cascades to guard scalar code. Our new contributions are as follows:

- we develop a novel IF-select transformation for vectorizing arbitrarily nested IF-statements based on the Open64 compiler for SW26010 processors;
- we suggest a new optimization technique for backend code generation of masked memory access LLVM-IR in LLVM compiler for targets without masked instructions, like IBM Power processors.

We integrate our first technique into the Open64 compiler, and we experimentally confirm its advantages using the SPEC CPU2006 benchmarks on a SW26010 processor used in the Sunway TaihuLight supercomputer. We integrate our second optimization technique into LLVM, and the experimental results on IBM Power processors demonstrate the efficacy of our approach.

**Acknowledgements** This research is supported by the Chinese Scholarship Council (CSC) scholarship, and by the German Federal Ministry of Education and Research (BMBF) in the Project HPC$^2$SE. Thanks are due to the National Supercomputing Center in Wuxi/China for providing access to the Sunway TaihuLight Supercomputer.

## References

1. Allen JR, Kennedy K, Porterfield C et al (1983) Conversion of control dependence to data dependence. In: Proceedings of the symposium on principles of programming languages (POPL), Austin, Texas, USA, pp 177–189. https://doi.org/10.1145/567067.567085
2. AMD (2012) Using the x86 Open64 compiler suite. For x86 Open64 version 4.5.2
3. Barton C, Tal A, Blainey B, Amaral JN (2005) Generalized index-set splitting. In: Bodik R (ed) Compiler construction. Springer, Berlin, pp 106–120
4. Bik AJC, Girkar M, Grey PM, Tian X (2002) Automatic intra-register vectorization for the Intel® architecture. Int J Parallel Program 30(2):65–98. https://doi.org/10.1023/A:1014230429447
5. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee S, Skadron K (2009) Rodinia: A benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, USA, pp 44–54. https://doi.org/10.1109/IISWC.2009.5306797
6. Cooper K, Torczon L (2011) Engineering a compiler. Elsevier, Amsterdam
7. Danalis A, Marin G, McCurdy C, Meredith JS, Roth PC, Spafford K, Tipparaju V, Vetter JS (2010) The scalable heterogeneous computing (shoc) benchmark suite. In: Proceedings of the 3rdWorkshop on General-Purpose Computation on Graphics Processing Units, ACM, pp 63–74. https://doi.org/10.1145/1735688.1735702
8. Free Software Foundation (2019) Using the GNU Compiler Collection (GCC). https://gcc.gnu.org/onlinedocs/gcc/. Accessed 24 May 2019
9. Fu H, Liao J, Yang J et al (2016) The Sunway TaihuLight supercomputer: system and applications. Sci China Inf Sci 59:1–16. https://doi.org/10.1007/s11432-016-5588-7
10. Henning JL (2006) SPEC CPU2006 benchmark descriptions. ACM SIGARCH Comput Archit News 34(4):1–17. https://doi.org/10.1145/1186736.1186737

11. Intel (2019) Intel 64 and IA-32 Architectures Optimization Reference Manual. Accessed May 2019
12. Intel (2017) Intel C++ Compiler Developer Guide and Reference. Version 18.0
13. Karrenberg R, Hack S (2011) Whole-function vectorization. In: Proceedings of the international symposium on code generation and optimization (CGO), Chamonix, France, pp 141–150. https://doi.org/10.1109/CGO.2011.5764682
14. Larsen S, Amarasinghe SP (2000) Exploiting superword level parallelism with multimedia instruction sets. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), Vancouver, BC, Canada, pp 145–156. https://doi.org/10.1145/358438.349320
15. Lattner C, Adve VS (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on code generation and optimization (CGO), San Jose, CA, USA, pp 75–88. https://doi.org/10.1109/CGO.2004.1281665
16. Lokuciejewski P, Gedikli F, Marwedel P (2009) Accelerating WCET-driven optimizations by the invariant path paradigm: a case study of loop unswitching. In: Proceedings of the 12th international workshop on software and compilers for embedded systems, SCOPES '09. ACM, New York, NY, USA, pp 11–20. http://dl.acm.org/citation.cfm?id=1543820.1543823
17. Moll S (2019) The Region Vectorizer (RV). https://github.com/cdl-saarland/rv. Accessed May 2019
18. Moll S, Hack S (2018) Partial control-flow linearization. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), New York, NY, USA. https://doi.org/10.1145/3192366.3192413
19. Pharr M, Mark WR (2012) ispc: a SPMD compiler for high-performance CPU programming. In: Innovative parallel computing (InPar). IEEE, pp 1–13. https://doi.org/10.1109/InPar.2012.6339601
20. Pohl A, Cosenza B, Juurlink BHH (2018) Control flow vectorization for ARM NEON. In: Proceedings of the 21st international workshop on software and compilers for embedded systems (SCOPES), May 28–30, 2018, Sankt Goar, Germany, pp 66–75. https://doi.org/10.1145/3207719.3207721
21. Shin J, Hall MW, Chame J (2005) Superword-level parallelism in the presence of control flow. In: Proceedings of the international symposium on code generation and optimization (CGO), San Jose, CA, USA, pp 165–175. https://doi.org/10.1109/cgo.2005.33
22. Shin J, Hall MW, Chame J (2009) Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. Microprocess Microsyst Embed Hardw Des 33(4):235–243. https://doi.org/10.1016/j.micpro.2009.02.002
23. Sreraman N, Govindarajan R (2000) A vectorizing compiler for multimedia extensions. Int J Parallel Program 28:363–400. https://doi.org/10.1023/A:1007559022013
24. Sujon MH, Whaley RC, Yi Q (2013) Vectorization past dependent branches through speculation. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT '13. IEEE Press, Piscataway, NJ, USA, pp 353–362. http://dl.acm.org/citation.cfm?id=2523721.2523769
25. Sun H, Fey F, Zhao J, Gorlatch S (2019) WCCV: Improving the vectorization of IF-statements with warp-coherent conditions. In: Proceedings of the 2018 International Conference on Supercomputing, ICS '19. ACM, New York, NY, USA, pp 319–329. https://doi.org/10.1145/3330345.3331059
26. Tanaka H, Ota Y, Matsumoto N, Hieda T, Takeuchi Y, Imai M (2010) A new compilation technique for SIMD code generation across basic block boundaries. In: 2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC), pp 101–106. https://doi.org/10.1109/ASPDAC.2010.5419911
27. Thomas J, Allen F, Cocke J (1971) A catalogue of optimizing transformations. Prentice-Hall, Englewood Cliffs
28. TOP500: https://www.top500.org/lists/2018/11/. Accessed 24 May 2019