Abstract—Modern architectures increasingly rely on SIMD vectorization to improve performance for floating point intensive scientific applications. However, existing compiler optimization techniques for automatic vectorization are inhibited by the presence of unknown control flow surrounding partially vectorizable computations. In this paper, we present a new approach, speculative vectorization, which speculates past dependent branches to aggressively vectorize computational paths that are expected to be taken frequently at runtime, while simply restarting the calculation using scalar instructions when the speculation fails. We have integrated our technique in an iterative optimizing compiler and have employed empirical tuning to select the profitable paths for speculation. When applied to optimize 9 floating-point benchmarks, our optimizing compiler has achieved up to 6.8X speedup for single precision and 3.4X for double precision kernels using AVX, while vectorizing some operations considered not vectorizable by prior techniques.

Index Terms—SIMD Vectorization, speculation, compiler optimization, iterative compilation, ATLAS, iFKO.

I. INTRODUCTION

With SIMD vector units becoming ubiquitous in modern microprocessors (e.g., x86 SSE/AVX, ARM NEON, POWERPC Altivec/VMX, among others), their effective utilization is critical to attaining a high level of performance for scientific applications. Most compilers, e.g., GNU gcc and Intel icc, can automatically vectorize instruction sequences when safe [22], [9], [18]. However, when instructions are embedded inside conditional branches, their vectorization is often inhibited due to the presence of unknown control flow. Existing research has exploited predicated execution of vectorized instructions [24], [23] to support SIMD vectorization of such instructions. However, without special hardware support, these techniques need to evaluate all the branches of a control flow before using special instructions to combine results from different branches, resulting in a significant amount of replicated computation whose results are never used.

Listing 1 illustrates this problem with a loop nest that includes partially vectorizable statements inside control flow branches. In particular, the statement s1 can be fully vectorized, s2 can be vectorized with predicated execution, and s3 cannot be vectorized due to loop-carried dependences. Figure 1 shows the control flow graph of these statements, where both s1 and s2 can be safely vectorized if Path-1 is taken at every vectorized iteration of the surrounding loop.

Listing 2 shows the result of vectorization using the predicated execution approach of Shin et al. [24]. Here s1, p1, and s2 are all vectorized, with the result of the vectorized p1 (vpT) serving as a mask in selecting the valid results of s2. Then, the predicate vector vpF is unpacked and used to

for (i=1; i<=n1024; i++)
{
  s1: a = A[i] * scal; /* vectorizable */
  p1: if (a <= MaxVal)
      s2: B[i] = A[i]; /* vectorizable */
      else
      s3: B[i] = B[i-1]; /* not vectorizable */
}

for (i=1; i<=n1024; i++)
{
  s1: Va[0:3] = A[i:i+3] * [scal,scal,scal,scal];
  p1: vcomp = Va[0:3] <= [MaxVal,MaxVal,MaxVal,MaxVal];
  s2: vpT, vpF = vpsel(vcomp);
  B[i:i+3] = select(B[i:i+3],A[i:i+3],vpT);
  s3: /* scalar part */
      [PF1,PF2,PF3,PF4] = UNPACK(vpF);
      if (PF1) B[i] = B[i-1];
      if (PF2) B[i+1] = B[i];
      if (PF3) B[i+2] = B[i+1];
      if (PF4) B[i+3] = B[i+2];
}

Listing 1: Example: vectorization in the presence of unknown control flow

Listing 2: SIMD Vectorization using predicated execution [24]
selectively evaluate the four unrolled instances of \( s_3 \). Note that \( s_2 \), now translated into two vectorized instructions, is always evaluated irrespective of the output of the predicates. Further, the unpacking of the predicate vector \( \text{vpT} \) could result in extra pipeline stall cycles within the CPU.

In this paper, we present a new approach, which speculates past dependent branches to enable aggressive vectorization of paths that are evaluated frequently at runtime. As illustrated in Listing 3, where the path composed of statements \( s_1 \) and \( s_2 \) is selected and speculatively parallelized, our approach checks the correctness of the speculation at a very early stage, and if the speculation fails, the alternative scalar iterations (\( s_3 \) in Listing 3) are evaluated instead. In addition to allowing the vectorization of routines that cannot be vectorized by existing techniques, our experimental results show that this speculative vectorization approach can outperform existing techniques when the control flow branches are strongly directional; that is, the vectorized path is frequently taken at runtime (e.g., kernels past dependent branches to enable aggressive vectorization of paths that are evaluated frequently at runtime). However, in situations where control flow paths are unpredictable (i.e., a random branch could be taken at any iteration), overly high misspeculation rate could result in our approach performing worse than the original code or code vectorized via predication. To ameliorate this limitation, we use an iterative compilation framework [26] to experiment with different path speculations so that the technique is applied only when beneficial for representative inputs.

We have implemented our speculative vectorization technique within iFKO [26], an iterative optimizing compiler that focuses on backend optimizations for computation-intensive floating point kernels which uses empirical tuning to automatically select the best performing transformations, and have used iFKO to perform SIMD vectorization for 9 floating point benchmarks with single and double precision variants. Our results show that up to 6.8X speedup for single precision and up to 3.4X speedup for double precision can be attained for these benchmarks in AVX through our speculative vectorization optimization. Our contributions include the following:

- We have integrated our technique within an iterative compiler and used empirical tuning techniques to automatically select the most profitable path to vectorize.
- We demonstrate the effectiveness of our techniques using a large number of floating point kernels, including some inherently scalar code, e.g., the sum of square computation for \( \text{nnm2} \) in the BLAS library [10], which could not be vectorized efficiently using existing techniques.

The remainder of the paper is organized as follows. Section II describes our algorithm for speculative vectorization. Section III summarizes our integration of the algorithm within the iFKO iterative optimizing compiler framework. Section IV describes our experimental methodology and the results obtained using speculative vectorization. Section V presents related work, and Section VI presents our conclusions.

### II. Speculative Vectorization

Our research aims to support aggressive SIMD vectorization of important loops even when their bodies contain complex control-flow and when the entire computation cannot be fully paralleled. Our solution effectively combines two classes of existing techniques, SIMD vectorization and path-based speculation, that have been highly successful in modern compilers.

To speculatively vectorize a loop, we first find all possible paths through the loop body. Analysis is then performed to determine which of these paths can be safely vectorized, and the set of safely vectorizable paths are returned to the search engine of iFKO, an iterative compilation framework for backend optimization. The search driver of iFKO invokes its optimizing compiler to experimentally vectorize iterations of statements along each path, measures the performance of the differently vectorized code, and finally selects the most profitable path to be vectorized for the original application. The following subsections present both the analysis and transformation steps in detail. The overall iFKO iterative compilation framework is then outlined in Section III-A.

#### A. Safety Analysis

Algorithm 1 outlines the main steps of our safety analysis algorithm, which takes a single input loop and returns a set of paths that are safe targets for speculative vectorization.

**Algorithm 1 speculative vectorization analysis**

```plaintext
func is_loop_vectorizable(loop)
    (1) if !(is_loop_form_vectorizable(loop))
        then return(\text{empty}); fi
    (2) paths = select_paths_to_speculate(loop);
    (3) foreach \( p \in \text{paths} \)
        do
            if !(is_path_vectorizable(p, loop))
                then paths = remove(p, paths); fi
        od
    return(vectorizable_paths)
end
```
In more detail the three steps of Algorithm 1 are:

1) **is_loop_form_vectorizable**: In step 1, we determine whether the input loop is in a form suitable for vectorization. In particular, we require that the loop must be regular (i.e., can be easily translated to a Fortran Do-style loop) and countable [16], where the number of iterations of the loop is known before entering the loop body, and all loop iterations can be counted using an integer index variable. If the input loop fails to satisfy this condition, it is considered unsafe to vectorize.

2) **select_paths_to_speculate**: In step 2, we determine which paths are candidates for speculative vectorization. Since the cost of finding all paths through a loop body could grow exponentially as the number of branches increases, our compiler takes an optional command-line argument that sets the maximum number of paths to consider for vectorization, and any remaining paths will not be considered once the threshold is exceeded. This threshold is implemented to ensure our optimization is never overwhelmed by overly complex control flow, which is not expected to happen often in practice. In particular, iFKO, the compiler infrastructure where we implemented our optimization, targets floating point kernels, which typically have fairly modest control flow complexity, and having too many paths to analyze almost never becomes a concern. Our compiler analyzes all paths for speculative vectorization for the benchmarks studied in our experimental evaluation.

3) **is_path_vectorizable**: In step 3, we determine the safety of vectorizing each path selected by step 2 using existing data flow and dependence analysis techniques [16], [18], [9], [22], [19], to categorize the vectorizability of variables and statements along the path. Any path that cannot be vectorized is removed from the existing collection of paths to be considered for speculation before the final result is returned.

In step 3, **is_path_vectorizable** classifies all variables inside a path into the following categories:

- **Invariant**: Variables that are used inside the speculated path but never modified within the path. During vectorization, these scalar invariants can simply be replicated inside vectors so that the same value is used across all vectorized iterations of the selected path.

- **Local or private**: Variables that are re-initialized at each iteration of the selected path before being used along the path. During vectorization, a vector needs to be allocated for each private variable to hold the value of the variable for each vectorized iteration.

- **Recurrent**: Variables that are modified along the path after being used in the current or previous iterations of the path. Special forms of recurrent variables, e.g., loop induction and reduction variables, can be vectorized in spite of their loop-carried cross iteration dependences. However, the existence of other more general forms of recurrent variables along the speculated path would prevent the path from being vectorized.

The above categorization is made using a data flow analysis approach similar to that taken in [19], except that only the speculated path is analyzed. For instance, a variable that is modified or recurrent within a loop can be invariant or private along a speculated path, thus allowing the path to be vectorized provided that proper recovery mechanisms are in place when the speculation fails. If a path contains a recurrence that is not induced by a loop induction or reduction variable, the path is deemed unsafe to vectorize and is removed from the set of paths to be considered for speculation.

To ensure each speculated path can be correctly vectorized by a later transformation step, our analysis additionally identifies all variables that belong to the following groups:

- **Live-out**: Variables that have been modified inside the vectorized path and are expected to be used after the loop terminates. For these variables, their values at the last iteration of the loop need to be copied or reduced back to scalar variables to ensure correct references to their values after the loop is complete.

- **Live-In**: Variables that are used along the vectorizable path before they are modified within the loop. These are live-in at the entry of the loop path. For these variables, their vector representations need to be initialized with correct values before entering the vectorized path. If the variable is a reduction variable, the first element of its vector representation is initialized with its scalar value before entering the vectorized path, and the rest of the elements are initialized with zero or one (e.g., 0 is used if the reduction operation is addition, and 1 is used for multiplication reductions). Otherwise, since the safety analysis considers this path to be vectorizable, the variable is not recurrent, and all entries of its vector can simply be initialized with the scalar value before entering the vectorized path.

Figure 2(a) shows an implementation of the SSQ kernel from the BLAS library in ATLAS [28]. The if-else conditional inside this loop has generated two alternative paths to consider, as shown in Fig. 2(b). Finally, Fig. 2(c) illustrates our variable classification analysis for both paths: Path-1, which includes the if-branch, modifies only one non-local variable, ssq, through reduction, and as a result is vectorizable. On the other hand, Path-2 modifies both ssq and scal in a complex recurrent fashion and thus cannot be vectorized. By selecting Path-1 to vectorize, our speculative algorithm can partially vectorize the given input loop even though Path-2 is not at all vectorizable. To the best of our knowledge, the loop in Fig. 2(a) is currently classified as **not vectorizable** by existing SIMD vectorization techniques.

**B. Structure of Generated Code**

Figure 3 shows the typical structure of the code generated by our vectorization transformation. Here all the statements in the original code are rearranged along two paths: the path taken when the speculative vectorization succeeds (i.e., the vectorized path), and the path(s) taken when the speculation fails (i.e., the scalar restart code). The vectorized path contains:
Fig. 2: Example: analyzing the Sum of Squares (SSQ) kernel

Variable Analysis:

Path-1 : B1 -> B2
Analysis on Path #1:
1. Local/Private variable: t0, ax
2. Invariant variable: scal, ABS
3. Reduction variable: sqs
4. LiveIn (at the entry): sqs, scal, ABS
5. Liveout (at the exit): sqs, scal

Result : Vectorizable

Path-2 : B1 -> B3
Analysis on Path #2:
6. Local/Private variable: t0, t1, ax
7. Invariant variable: ABS
8. Recurrence variable: sqs, scal
9. LiveIn (at the entry): sqs, scal, ABS
10. Liveout (at the exit): sqs, scal

Result : Not Vectorizable

C. Applying The Transformation

Listing 4 shows an example of the speculatively vectorized loop from the original code in Fig. 2(a). The analysis of the two paths through this loop are shown in Fig. 2(c), where Path-1 has been selected for speculative vectorization. Fig. 4(a) shows the initial control-flow graph for this loop, and (b)-(d) illustrate the intermediate results of our vectorization transformation. In our implementation, the speculative vectorization transformation is applied through the following five steps:

1. Speculated path formation: This step modifies the control flow of the loop body so that each conditional branch inside the speculated path (spath) is a potential exit from the spath to the unvectorized code, and all blocks that are not in the chosen path are relocated to a separate region (which will be converted to scalar restart code in step 3). This code reorganization leaves the chosen spath contiguous in instruction memory with the loop, increasing its spatial locality and decreasing the probability of branch mispredicts within
the path. In order to make the spath instructions contiguous, it is necessary to reverse the branch conditionals\(^1\) whose fallthrough and goto targets are swapped by this transformation, resulting in the modified CFG shown in Fig. 4(b). Note that blocks B2 and B3 have changed position from Fig. 4(a).

### 2. Vectorization alignment and cleanup

In this step, we perform possible loop peeling in order to align vector memory access [16], [13], as well as creating a cleanup loop to handle loop iterations that are not a multiple of the vector length [1], [2], [26]. This step is not particular to speculative vectorization, and for simplicity this cleanup/alignment code is generally omitted from our figures.

### 3. Scalar Restart Generation

This step uses the current scalar loop to generate the scalar restart code. As shown

\(^1\)reversing conditionals can complicate NaN handling; in our framework, like many compilers, this transformation is allowed

\(^2\)Speculation is proven correct after the last conditional exit from the spath.

in Fig. 3, the scalar restart restores any possibly modified recurrent variables, reduces the vector values to scalar values, and then recomputes all speculated iterations using a scalar loop, before doing scalar-to-vector initialization, and then branching back to the loop update block. At this point, the scalar restart code is complete, but the spath does not yet have the branch target information to reach it, which is handled in the next step, and the spath is not yet vectorized, which is done as the final step. In Listing 4, the scalar restart code is shown at lines 19-48: Here a single reduction variable, ssq, needs to have its scalar value restored from vectorized evaluations (line 24). Its scalar evaluation result is then later transferred back to its vector variable at line 47. A variable scal is modified along the scalar path at line 37 and used in the speculatively vectorized path at line 15. Therefore, its value is transferred to a vector variable at line 48 before executing the vector loop update.

### 4. Branch target repair and non-spath block removal

This step updates all conditional branch targets out of the spath with the label of the scalar restart code generated in the previous step. Since they are now handled by our scalar restart code, the original non-speculated path(s) from the loop are no longer referenced anywhere in the code and are therefore removed. In our example, this results in the deletion of block B3, giving rise to the CFG shown in Fig. 4(c). At this point the control flow of the transformed code is correct, but the instructions along spath have not yet been vectorized, which is done by the final step.

### 5. spath Vectorization

Finally, this last step vectorizes all statements along the selected spath and then adds the necessary vector-prologue, vector-backup, and vector-epilogue, as outlined in Fig. 3. In particular, all recurrent variables that may be modified before the last scalar restart exit\(^2\) are backed up before any vectorized evaluation. In order for our speculation to be true, each conditional branch along spath must take the fall-through direction for all speculated iterations. Therefore, we replace each original branch comparison with a vector comparison/test that exits to the scalar restart code if any component of the comparison failed to match our speculated result. The final CFG, including the loop cleanup, is shown in Fig. 4(d). Listing 4 shows a simplified pseudo-code for the vectorized loop of our SSQ example (excludes loop peeling and loop cleanup).

### D. Correctness And Generality

The main novelty of our speculative vectorization algorithm lies in the insight that when branches within a loop are strongly directional, that is, when consecutive iterations of the loop are expected to take a speculated control-flow path most of the time, SIMD vectorization can be applied to aggressively parallelize the path, with the other paths given lower priority. A similar path-based formulation has been used in trace scheduling [3], the de facto instruction scheduling algorithm widely adopted by modern compilers. However, such formulation has
yet to be extended to other backend compiler optimizations beyond instruction scheduling. As far as we know, our work is the first that formulated SIMD vectorization using path-based optimization strategies.

Since our work essentially extends existing SIMD vectorization algorithms [16], [18], [9] to support control-flow path speculation and recovery, the algorithm is correct as long as the control-flow transformations are correctly performed, all the variables mistakenly modified by the speculated path can be correctly recovered, and the spath code branches correctly to the scalar restart code when misspeculation is detected. Our current implementation supports only speculative vectorization of a single path within a given loop, and the vectorization is disabled when the path contains memory references or variables that cannot be precisely modeled.

III. INTEGRATION WITHIN IFKO

We have implemented our speculative vectorization optimization, together with several other transformations to help evaluate its effectiveness, within iFKO [26], an iterative backend compiler with an emphasis on optimizing the performance of floating-point intensive computational kernels. Section III-A and III-B provide an overview of the iFKO tuning framework and the new capabilities that we have added. Section III-C discusses empirical tuning strategies we have adopted within iFKO to automatically find the fastest available vectorization method for each input kernel.

A. Overview of iFKO

Figure 5 shows the overall structure of the iFKO [29], [26] (iterative Floating Point Kernel Optimizer) compilation framework. The iterative compiler is composed of two components, a set of search drivers that search the optimization space, and a specialized compiler called FKO that performs analysis (to determine legality of transforms as usual, but in an iterative compiler, also to bound the search space), and makes all required transformations. Two things must be supplied to iFKO by the user: (1) the routine to be compiled (in a HIL similar to a restricted C), and (2) a context sensitive timer [27] for the kernel being tuned. We extended ATLAS’s [28] timing framework in order to autotune the surveyed benchmarks.

In iFKO, optimizations are split into two classes. Fundamental transformations are optimizations that are empirically tuned during the timing process, while repeatable transformations are optimizations that are repeatedly applied in series to a scope of code while they are successfully improving the code. Fundamental transforms usually have a parameter that is searched during the tuning phase. In the simplest case, the search is whether or not to apply an optimization, since it only sometimes leads to faster code. But often an optimization itself is parameterized, as in loop unrolling, where the search will find the best-performing unrolling factor in a large range. Examples of parameterized fundamental transformations include loop unrolling, prefetch distance, and accumulator expansion (see [29] for the original list of 7 fundamental transforms). Most of the repeatable transformations in iFKO are centered around optimizing register usage, see [26] for full details.

B. Extending iFKO Fundamental Transformations

In the original iFKO, SIMD vectorization was a fundamental operation with only a yes/no parameterization, as vectorization can produce a slowdown for some operations and machines. The compiler supported simple loop-based vectorization, which is enabled when the dependence distance (control & data) is greater than the vector length of the underlying architecture. We will refer to this original vectorization method as NHV, for No Hazard Vectorization.

The inability of the original iFKO to apply NHV in the face of control hazards prevented it from vectorizing all of the Level 1 BLAS [29]. For this work, we added five new fundamental transformations to support vectorization past branches (some of these new optimizations help even in scalar code, as described below). These transformations are all searched by iFKO, so the best performing optimizations will be automatically selected for the user. In order to compare
them in this paper, we have overridden the search using flags to require certain transformations be applied instead of searched.

We have added two new fundamental transformations that do not themselves perform vectorization, but rather transform the scalar code so that control hazards are removed, with the result that the loop can then be vectorized by NHV:

1) **MMR (Max/Min Reduction):** Automatically detects simple if-conditions that serve only to compute a max or min over a sequence of values. Once found, it replaces the entire branch with the assembly MAX/MIN instruction. When MMR alone is sufficient to allow vectorization using NHV, we refer to this series of transformations leading to vectorization as VMMR.

2) **RC (Redundant Computation):** Seeks to eliminate conditional branches by replicating computations along different branches and then selecting the proper values in a fashion similar to [24]. When RC alone is sufficient to allow vectorization using NHV, we refer to this series of transformations leading to vectorization as VRC.

Note that in this paper we never need to apply both MMR and RC in order to vectorize, so this case is not discussed.

Our speculative vectorization implementation is supported by the following additional fundamental transformations:

3) **FPC (Frequent Path Coalescing):** Rearranges the control flow within a loop so a given path becomes a straight-line sequence of code intermixed with conditional exit jumps out of the path.

4) **SV (Speculative Vectorization):** If the loop targeted for vectorization has non-loop branches, examine all possible paths through the loop, and discover which are vectorizable. Our present algorithm will vectorize only one path through the loop (this simplifies our analysis & scalar restart code, but it should be possible to vectorize all legal paths with improved compilation phases). Use FPC to make the target path fall-through, and then vectorize it. All other paths are handled by scalar code.

iFKO already has a fundamental optimization called UR, which does straightforward loop unrolling. In this type of unrolling, the loop body is simply replicated as many times as requested, while avoiding moving pointers and changing loop control between unrolled iterations. We have implemented a second version of unrolling that can be used in conjunction with the existing one, so that the best performing unrolling optimization can be selected based on timing results of the optimized code.

5) **OSUR (Over-Speculation loop UnRolling):** in this type of unrolling, we speculate the path to a non-unit multiple of the vector length and inline multiple vectors of computation. This will usually pay off only for branches with very strong directional preferences, but its advantage over normal unrolling is that the overhead of speculation checking is more completely amortized by the increased speculation length. During the search, we will time OSUR & UR alone, as well as combinations of the two whenever we are tuning an SV-vectorized loop.

### C. Optimization Tuning

FKO returns to the search driver a list of all possible paths through the loop. This information is then used in the following process to find the best optimized code:

- If the number of paths is one, and it is vectorizable, time both scalar code and code vectorized by NHV, and choose the best.
- If there are multiple paths through the code, choose the best performing code among:
  - Scalar code
  - VMMR (if applicable)
  - VRC (if applicable)
  - SV: For each path that is vectorizable, apply SV and time it, and return the best-performing code. Note that the user’s timer (and its associated training data) can have a profound impact, as highlighted in Section IV-B.

### IV. Experiments

To validate the effectiveness of our speculative vectorization technique and the performance benefit of integrating it within an iterative optimizing compiler, we have applied the techniques to optimize 9 benchmarks, summarized in Table I, with both single precision and double precision versions for each benchmark, on two machines using Intel and AMD processors respectively. The specification of the machines are listed in Table II. All timings utilize data chosen to fit the operands in the L2-cache, while overflowing the L1-cache; sin and all irkamax kernels utilize an 8,000 vector length input; all other kernels use a 16,000-element input to satisfy the same cache constraints. For all kernels except sin and cos, the input values use random numbers in the range [-0.5, 0.5]. For sin & cos, however, this would give our technique a strong advantage and is probably not realistic (see Section IV-B for further details). For these two kernels, we instead generate the input by passing the random values between $[0, 2\pi]$ to the wrapper functions from glib that call these kernels; This essentially guarantees that all paths in the kernels are executed, and thus represents the worst case for our technique.

**Table I: Benchmarks used for experiments**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description and Library</th>
<th>Input Data and Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMAX</td>
<td>Absolute max value search</td>
<td>rand([-0.5,0.5]), in-L2</td>
</tr>
<tr>
<td>TAMAX</td>
<td>index of absolute max, blas</td>
<td>rand([-0.5,0.5]), in-L2</td>
</tr>
<tr>
<td>SSQ</td>
<td>ssq for norm2, blas</td>
<td>rand([-0.5,0.5]), in-L2</td>
</tr>
<tr>
<td>ASUM</td>
<td>Absolute sum, blas</td>
<td>rand([-0.5,0.5]), in-L2</td>
</tr>
<tr>
<td>IRK1AMAX</td>
<td>Panel factorization of LU, AT-LAS</td>
<td>rand([-0.5,0.5]), in-L2</td>
</tr>
<tr>
<td>IRK2AMAX</td>
<td>Panel factorization of LU, AT-LAS</td>
<td>rand([-0.5,0.5]), in-L2</td>
</tr>
<tr>
<td>IRK3AMAX</td>
<td>Panel factorization of LU, AT-LAS</td>
<td>rand([-0.5,0.5]), in-L2</td>
</tr>
<tr>
<td>KERNEL_SIN</td>
<td>Kernel for sine of glibc (version:2.4.2.15)</td>
<td>rand([0, 2\pi]) x on sin(); use realistic input of kernel_sin using _kernel_rem_pio2(), in-L2</td>
</tr>
<tr>
<td>KERNEL_COS</td>
<td>Kernel for cosine of glibc (version:2.4.2.15)</td>
<td>rand([0, 2\pi]) x on cos(); use realistic input of kernel_cos using _kernel_rem_pio2(), in-L2</td>
</tr>
</tbody>
</table>
A. Effectiveness of Speculative Vectorization

Figure 6 shows the speedup of speculative vectorization over the scalar code for Intel and the AMD machine. Both machines are using AVX, with a vector length of 8 (4) in single (double) precision. Note that both the scalar and vector versions have been empirically tuned by iFKO, so our scalar code represents the best possible case without vectorization (i.e., it is not a naive unoptimized baseline).

The first point to notice from the results is that the performance benefit of applying our vectorization technique on the Intel machine is almost twice of that on the AMD. Getting peak AVX performance from the AMD Dozer is complicated by the fact that on the backend the 256-bit AVX operations are split into two separate 128-bit operations, unlike on the Intel which has true 256-bits FPU's. AMD's more complex AVX handling tends to complicate scheduling on a machine that is already weak in that area, and it is also sometimes required to mix SSE and AVX instructions to maximize performance.

We expect good performance from SV only when the vectorized path is preferred. This is certainly the case for kernels such as `amax`, `iamax`, `nrm2`, `irk1amax`, `irk2amax`, and `ir3amax`; all benefited significantly from speculative vectorization. For `asum`, SV actually causes a slowdown. Note that since our compiler can automatically select the best optimized code, SV would not be selected to optimize `asum` by our compiler.

Similar observations can be made for `cos` and `sin`, where multiple paths are selected based on the input data range. Single precision `cos` experiences a slight slowdown on the Intel machine, and other `cos` and `sin` results show very modest speedup. Since our speculation is almost always wrong on these kernels, the fact that we achieve any speedup at all is a measure of how low the overhead of our scalar restart code is. Of course, when iFKO is allowed to fully autotune codes such as this, the tuning framework will choose an alternative vectorization strategy (e.g., redundant computation) or not vectorize the code at all.

B. Comparing with other Vectorization Techniques

The main strength of speculative vectorization is that it can be used in cases where the known techniques cannot be applied. In particular, if there are multiple paths through the loop, only some of which can be successfully vectorized, SV is the only technique capable of realizing vector speeds. The `nrm2` performance shown in Fig. 6 is an example where SV allowed us to get impressive speedups when no other vectorization can be applied.

However, many kernels can be vectorized in different ways, and a compiler can always select the most promising approach based on characteristics of the input application. A reasonable heuristic can be constructed using the following line of reasoning: (1) If branches are used only for max or min, then replacing them with machine native MAX/MIN instructions (VMMR). (2) If all paths are vectorizable, and the cost of computing all sides of the branches is low, then replicate all branches to enable vectorization (VRC). (3) If a vectorizable path is strongly directional then consider speculative vectorization (SV).

Figure 7 shows the performance of all three vectorization methods on `amax` for the Intel Corei2. This computation is inexpensive and strongly directional, therefore a good case for both SV and VRC. We see they are both fairly competitive with VMMR, with SV performing slightly better than VRC.
specifically chose data in the range of $[0,2\pi]$ using scalar code tuned and timed for data in range $[-0.5, 0.5]$ (scal.5), Speculative Vectorization tuned and timed in range $[0, 2\pi]$ (SV2pi), and range $[-0.5, 0.5]$ (SV.5), and Vectorized Redundant Computation tuned and timed in range $[0, 2\pi]$ (VRC2pi) and range $[-0.5, 0.5]$ (VRC.5) on this machine (this essentially means that our scalar restart overhead is lower than the overhead of doing the vector compare and select). In general, we would expect that VMMR should win whenever it can be used, while the VRC and SV performance ratio will vary depending on how predictable the path is, and how much work must be performed redundantly.

Figure 8 compares the different vectorization methods using a $\sin$, and shows how path selection can have large effects on speculative vectorization. For this benchmark in Fig. 6 we specifically chose data in the range of $[0, 2\pi]$ which exercises all the paths in the $\sin$ kernel; this prevents SV from producing much speedup on the AMD system. Here we instead tune and time the code using our usual range of random inputs between $[-.5, .5]$. VRC is unaffected, since it is always executing code from all paths. However, this has a profound affect on SV, since it results in a particular path dominating the kernel calls made by the full $\sin$ function. As a result it goes from showing almost no speedup, to greater speedup than any other method, as SV does not perform any redundant computation and only occasionally needs to do scalar restart. Note that these speedups are inflated because we are using the speed achieved in the range of $[0, 2\pi]$ as our denominator. Frequent path coalescing and related optimizations improve even the scalar code by almost a factor of 2 when we specifically tune for the $[-0.5, 0.5]$ data range. This input sensitivity of SV is both a hazard and a meaningful opportunity for application-specific tuning for applications with known typical ranges on their data.

V. RELATED WORK

The ubiquitous support of short vector operations in modern architectures has made SIMD vectorization one of the most important backend optimizations in modern compilers [14], [25], [2], [12], [24], [7], [17]. Bik et al. used bit masking to combine different values generated from different branches of if-else branches [2]. Shin, Hall, and Chame [24] managed dynamic control flow inside vectorized code through predicated execution of vectorized instructions and have implemented their schemes using mask and select vector operations. The technique was later improved to bypass some of the redundant vector computations for complex nested control flows [23]. Karrenberg et al. [11] presented a similar approach but introduced the mask and select operations in the SSA form to handle arbitrary control flow graphs. Our work also aims to enhance the effectiveness of automatic vectorization in the presence of complex control flow. Our techniques, however, focus on speculatively vectorizing strongly biased control-flow paths that are expected to be taken frequently at runtime. Our vectorization algorithm is based on existing loop-based vectorization techniques [2], [25], [7], but the path speculation strategy can be used to enhance superword-level vectorization frameworks [12] in a similar fashion.

Speculation is an approach commonly used in compilers when facing unknown control or data flow that prevent effective optimization [6], [15], e.g., instruction scheduling [6], [8] and thread-level parallelization [21], [4], [5]. Pajuelo et al. [20] proposed micro architecture extension to apply vectorization speculatively. To the best of our knowledge, our work is the first that uses path-based speculation to enhance the effectiveness of SIMD vectorization within compilers.

VI. CONCLUSION AND FUTURE WORK

This paper presents a new technique, speculative vectorization, which extends existing SIMD vectorization techniques to aggressively parallelize statements embedded inside complex control flow by speculating past dependent branches and selectively vectorizing paths that are expected to be taken frequently at runtime. We have implemented our technique inside the iterative backend optimizing compiler, iFKO, and have applied the path-based speculative vectorization approach to optimize 9 floating point kernel benchmarks. Our results show that up to 6.8X speedup for single precision and up to 3.4X speedup for double precision can be attained for these benchmarks in AVX through our speculative vectorization optimization. Our formulation allows partial vectorization of computations in the presence of complex control flow beyond what has been supported by existing known SIMD vectorization techniques.

Our speculation approach is complimentary and can be applied to enhance the effectiveness of most existing SIMD vectorization techniques. In future work, we will investigate applying path speculation in conjunction with known techniques. For instance, in kernels with multiple branches inside
the loop, it may make sense to eliminate some branches with redundant computation, while speculating past others, and this may lead to much greater speedups than either technique can achieve when applied in isolation. A related idea is to speculate more than one path for kernels possessing more than one vectorizable path.

As vector lengths continue to grow, it may become increasingly unlikely that a branch will go in the same direction for the entire vector length for many kernels (branches such as underflow/overflow guards should be unaffected by increasing length). For kernels where increasing vector lengths are problematic, we will need to investigate underspeculation, where we speculate to only some fraction of the vector length. This is a classic tradeoff where increased speculation accuracy reduces peak SIMD performance; by using empirical tuning we can find the most effective tradeoff, whether that is full, under-, or over-speculation.

Another technique that should be complementary with speculative vectorization is an adaptation of loop specialization, where we maintain the original scalar loop in the code along with the speculatively vectorized loop, and, if at runtime we detect too many jumps to the scalar cleanup code, we switch to the unvectorized code for the rest of the computation. The only thing that we would need to add to our framework to support this is scalar restart counting and some generalization of our loop specialization code, which should be straightforward.

ACKNOWLEDGMENT

This work was supported in part by the NSF CAREER grant # OCI-1149303, CCF-1261778, and CCF-1261811, and by the Department of Energy grant # DE-SC0001770. We would like to thank the reviewers for their suggestions, which much improved this paper.

REFERENCES