Combining Run-time Checks and Compile-time Analysis to Improve Control Flow Auto-Vectorization

Bangtian Liu*
University of Toronto, Canada
bangtian@cs.toronto.edu

Avery Laird*
University of Toronto, Canada
alaird@cs.toronto.edu

Wai Hung Tsang
IBM, Canada
whitney.uwaterloo@gmail.com

Bardia Mahjour
IBM, Canada
bmahjour@ca.ibm.com

Maryam Mehri Dehnavi
University of Toronto, Canada
mmehride@cs.toronto.edu

ABSTRACT

SIMD (Single Instruction Multiple Data) instructions apply the same operation to multiple elements simultaneously. Compilers transform codes to exploit SIMD instructions through auto-vectorization. Control flow can lead to challenges for auto-vectorization tools because compilers conservatively assume branches are divergent. However, it is common that all SIMD lanes follow the same control-path at run-time, a property we call *dynamic uniformity*. In this paper, we present VecRC (an auto-vectorizer with run-time checks), a novel compile-time technique that uses run-time checks to test for dynamically uniform control flows. Under the assumption of dynamic uniformity, we perform several compile-time analyses that improve control flow auto-vectorization vs state-of-the-art approaches. VecRC leverages dynamic uniformity to vectorize loops with control-dependent loop-carried dependences. Existing strategies use speculation to optimistically execute vector code, and must correct any incorrect computation due to violated run-time assumptions. VecRC performs compile-time analysis based on uniformity to support such dependences without the overhead of speculation. We propose a probability-based cost model to predict the profitability of run-time checks to avoid the specialized profiling or expensive auto-tuning required in existing methods. VecRC is evaluated in LLVM on a diverse range of benchmarks including SPEC2017, NPB, Parboil, TSVC, and Rodinia on Intel Skylake and IBM Power 9 architectures. On the Skylake architecture, geometric mean speedups of 1.31x, 1.20x, 1.19x, and 1.06x over Region Vectorizer, GCC, Clang, and ICC are obtained with VecRC on real benchmark code.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**.

KEYWORDS

Vectorization, SIMD, Data Dependence, Control Flow

1 INTRODUCTION

SIMD instructions (Single Instruction Multiple Data) improve the performance of parallel operations by applying the same operation to multiple elements simultaneously (also called *vectorization*). Well-known compilers such as GCC, Clang, and ICC use loop vectorization [29] and Superword Level Parallelism (SLP) [14] to take advantage of SIMD instructions. However, control flow (such as if-statements) can cause these techniques to generate inefficient code, because it is unknown at compile-time which control paths will execute. Additionally, if a data dependence exists, then conventional loop auto-vectorization must be disabled entirely. Although such codes can often be manually vectorized using intrinsics (a source code mechanism to use SIMD instructions explicitly), the manual effort is tedious, time-consuming, and often does not lead to the best attainable performance. Auto-vectorization provided by compilers is one feasible solution to allow generality and architecture portability, however control flow and data dependence both limit current auto-vectorization methods.

Control flow is typically auto-vectorized using *if-conversion* [1]. Under if-conversion, branches are removed and control-dependent instructions are guarded by predication techniques (such as masked instructions supported by instruction set architectures) to prevent illegal computation (see Section 2.1 for more background). For example, a conditional load might load from a null address unless the condition is true; this conditional load is converted to a masked load that takes as extra input a bit-mask. The bit-mask indicates for which lanes the load should be performed. While if-conversion allows loops with control flow to be vectorized, it has two major downsides. First, software predication has overhead [9]. Second, if the condition of the branch is often false during run-time, then most SIMD lanes will be unused, resulting in redundant computation. Similarly, if the branch condition is often true, then all lanes might frequently be active, obviating the need for software predication. Prior work has used static analysis to detect these worst cases (always true or always false conditions) and avoid applying if-conversion for a class of branch conditions defined as *uniform*. A branch is uniform when static analysis can determine that a condition has the same value for all SIMD lanes [20]. We use the term *dynamic* uniformity to describe the case when uniformity occurs at run-time but cannot be proven statically.

Dynamic uniformity has been shown to exist in many different applications [32]. Several existing methods propose run-time techniques to check branch conditions for dynamic uniformity, allowing more cases of uniformity to be detected [21, 28, 32]. In these cases, benefit can be achieved either by skipping regions when all lanes are inactive, or by eliminating software predication when all lanes are active. If dynamic uniformity occurs frequently during execution, the run-time check overhead is likely to be amortized by the benefits of each technique. However, the actual frequency of dynamic uniformity for a specific branch condition is unknown at

*Both authors contributed equally to this research.

compile-time. Therefore, these methods are limited when estimating the tradeoff between the additional overhead of run-time checks and the benefit from skipping or removing predication. As a result, existing techniques either rely on expensive autotuning, highly specialized profiling, or manual annotation to select profitable loops [26, 28, 32]. Additionally, prior works focus only on the benefits of skipping or removing predication, and do not take advantage of dynamic uniformity to perform additional compile-time analysis.

Another challenge for control-flow auto-vectorization is how to deal with control flow affected by dynamic loop-carried dependences. Several run-time techniques are proposed and aim to solve this problem [3, 31]. One prior work enables auto-vectorization by detecting control-paths without dependence, speculatively executing those paths as vector code, and "rolling back" operations when execution deviates from that path [31]. FlexVec proposes a hardware solution to speculatively execute vector code using a new instruction set architecture that detects dynamic loop-carried dependence, then correcting any computation affected by violated dependences [3]. However, due to the cost of either rolling back or correcting speculated operations, both techniques can only achieve benefit when loop-carried dependences occur infrequently. The control-path vectorization technique must choose from different candidate paths, and relies on autotuning to find the optimal vectorized path [31]. FlexVec performs specialized profiling to determine certain parameters (such as the number of loop-carried dependences that occur) that are used by heuristics to select profitable loops.

Our proposed compiler approach, implemented in VecRC (an auto-vectorizer with run-time checks), improves control flow auto-vectorization through the following contributions:

- We propose to perform compile-time analyses, such as dependence analysis, under the assumption of dynamic uniformity detected at run-time to relax some constraints on vectorization and improve control flow vectorization in advance.
- A comprehensive probability-based cost model to determine when run-time checks should be enabled by evaluating both the benefits and overhead caused by run-time checks.
- An implementation of VecRC in LLVM and an end-to-end evaluation on the Test Suite for Vectorizing Compilers (TSVC) and standard benchmarks: SPEC2017, Rodinia and NAS parallel benchmark suites. The geometric mean speedup is 1.21x and 1.19x over Clang on Power 9 and Skylake respectively.

2 MOTIVATION

In this section, we first present the existing techniques for control-flow vectorization and then use one running example to describe the approach implemented in VecRC.

2.1 Background

Control Flow Graph: A control flow graph (CFG) $G = (V, E)$ contains basic blocks $b \in V$, where V is a set of basic blocks, and directed control-flow edges $(b, s) \in E$, where E is a set of edges between basic blocks in V. A basic block represents a series of instructions executed sequentially. Branch instructions $br \in B$

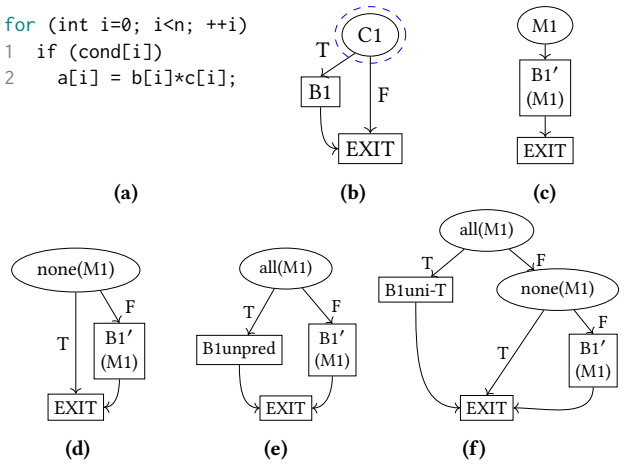


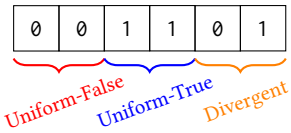
Figure 1: (1b) Control flow graph of loop body in (1a). (1c) shows if-conversion applied to the CFG in (1b), and instructions in block B1 are predicated by the mask M1 generated from the condition C1. (1d) With run-time check to detect uniform-false predicates. (1e) With run-time check to detect-uniform true predicates; block B1unpred has no predication. (1f) With VecRC run-time checks to detect both types of uniformity.

have one successor (if the branch is unconditional) or multiple successors (if the branch is conditional). Every CFG has one *entry* block, which has no predecessors, and at least one *exit* block, which has no successors [4]. The notation $b \rightarrow s$ denotes that there is a path from block b to s through G. A block $b \in V$ *dominates* $s \in V$ if b occurs in all paths to s; in other words, b must execute for s to execute. Conversely, block s *postdominates* b if s occurs in all paths $b \rightarrow e$, where e is an exit block [6]. A block s in G is *control-dependent* on a block b if there is a path $p : b \rightarrow s$ such that s postdominates every node after b on p, and s does not postdominate b. Informally, b is a branch with two outgoing paths: s appears along one path, but not the other. For each block $k \in V$, the notation $cblocks(k) \in V$ denotes the set of all blocks on which k is control-dependent. [4, 8]. For example, $cblocks(B1) = \{C1\}$ in Figure 1b, as shown by the blue dashed circle. When loops contain control flow, control dependence occurs in the CFG. Vectorization techniques for this code pattern, such as if-conversion, rely on the CFG to perform their transformations.

Compile-time Technique for Control-flow Vectorization based on Predication: If-conversion identifies control-dependent blocks and, for each such block, computes a block predicate (or *mask*) that represents which lanes should be active. In the remainder of the paper, we refer to control-dependent blocks and predicated blocks interchangeably. Figure 1b shows the CFG of the loop body from Figure 1a, and Figure 1c shows CFG 1b after applying if-conversion. In the original loop, line 2 (corresponding to B1 in 1b) can only execute if `cond[i]` is true. However, once the CFG 1b is if-converted, B1′ executes on each iteration; to prevent illegal computation, if-conversion predicates the loads and stores in B1′ using the block predicate M1. Instructions are placed at the beginning of a block that compute its predicate, using strategies such as those proposed by Park et al. [4].

```
for (int i=0; i<n; ++i) {
  if (cond[i]) // without dependence
    b[i] = a[i] + b[i];
  else { // with dependence
    j++;
    d[j] = c[i] * b[i];
  }
}
```

(a) Loop with control flow and control-dependent data dependence.



(d) Example elements in the cond array.

```
for (int i=0; i<n/2*2; i+=2) {
  v_cond = <cond[i], cond[i+1]>;
  /* uniform-true */
  if (all(v_cond)) {
    v_b = v_load(&b[i]);
    v_a = v_load(&a[i]);
    v_store(&b[i], v_a + v_b);
  } else /* scalar restart */
    for (int k=0; k<2; ++k)
      if (cond[i+k])
        b[i+k] = a[i+k] + b[i+k];
      else {
        j++;
        d[j] = c[i+k] * b[i+k];
      };
}
```

(b) Prior work [31]: vectorize the control-path without dependence.

```
for (int i=0; i<n/2*2; i+=2) {
  cond_v = <cond[i], cond[i+1]>;
  v_b = v_load(&b[i]);
4 if (all(cond_v)) {
    v_a = v_load(&a[i]);
    v_store(&b[i], v_a + v_b);
7 } else if (none(cond_v)) {
    v_c = v_load(&c[i]);
    v_d = v_c * v_b;
10  j++;
    v_store(&d[j], v_d);
12  j++;
  } else { /* Divergent path. */ }


}
```

(c) VecRC: vectorize the control-flow with dependence.

Figure 2: Comparision of auto-vectorization techniques for control-flow with dynamic uniformity (vectorization factor is 2). The loop example 2a contains a branch whose condition depends on the elements of `cond` (example elements shown in 2d). The values stored in `cond` result in two instances of dynamic uniformity. Elements 0 and 1 are both `0` and all lanes will be inactive (uniform-false). Elements 2 and 3 are both `1` and all lanes will be active (uniform-true). Elements 4 and 5 have different values and no uniformity exists (the divergent case). Loop 2b shows loop 2a after applying the run-time check technique proposed by Sujon et al [31]. 2c shows loop 2a after applying the VecRC run-time technique, which supports the loop-carried dependence from `j++` at else branch. VecRC's scalar evolution analysis recognizes that `j` can be described as a chain of recurrences under uniformity [2] and uses this insight to vectorize the store to `d` (shown in 2c).

Run-time Techniques for Control-flow Vectorization: While if-conversion is an entirely compile-time technique, each block predicate (for example, *M1*) may change dynamically during execution. Prior works that use compile-time divergence analysis [5, 17, 20] to determine whether a branch condition is statically uniform cannot analyze the behaviour of such data-dependent block predicates at run-time. Existing run-time techniques address the limitations of compile-time analysis by dynamically detecting whether a block predicate is uniform-true (all lanes active) or uniform-false (all lanes inactive) during execution [28, 32]. When a block predicate is uniform-false, redundant computation can be skipped using a run-time check proposed by earlier work [28], shown in Figure 1d as *none(M1)*. If the block predicate is uniform-true then an unpredicated version of the block can be executed to avoid predication overhead, using the *all(M1)* run-time check shown in 1e) [32]. Figure 1f shows how VecRC uses both *all()* and *none()* run-time checks to detect uniform-true and uniform-false predicates.

2.2 Motivating Example

Example Loop. Figure 2a shows an example loop that exhibits one code pattern VecRC focuses on. The branch condition depends on the data in `cond` (shown Figure 2d) which cannot be analyzed at compile-time. Additionally, the loop contains a control-dependent loop-carried dependence (statement `j++`). The control-flow in the loop presents one obstacle to auto-vectorization. Additionally, the loop-carried dependence from statement `j++` disables if-conversion and also prevents the compiler from determining the access pattern on array `d`.

Run-time Techniques to Test Dynamic Uniformity. Figure 2b shows an example of the run-time technique proposed by Sujon et al. to auto-vectorize loops with control-flow and control-dependent loop-carried dependences [31]. If a control-path does not contain any dependence, and is known to execute for consecutive iterations equal to (or greater than) the vectorization width, then such a control-path can safely be executed as vector code. This strategy uses run-time checks (*all(M1)* in Figure 2a) to detect whether such control-paths have dynamically uniform conditions, in other words, whether they will be executed for the required consecutive iterations. As shown in Figure 2c, VecRC detects both uniform-true and uniform-false predicates.

Dependence Analysis based on Dynamic Uniformity. Figure 2a shows an example loop with a control-dependent loop-carried dependence (`j++`) that cannot be vectorized. Since the `j++` statement is known to occur in all iterations of the uniform-false path, other non-dependent instructions can be vectorized if `j++` is properly scheduled (lines 10 and 12 of Figure 2c). Because this transformation is only valid when the condition is uniform-false, a run-time check guards the vectorized code (line 7). Similarly, another run-time check detects the uniform-true case when vector code can be executed without predication (line 4). This example shows just one form of dependence addressed by VecRC; further details are explored in Section 3.1.

Scalar Evolution Analysis based on Dynamic Uniformity. Normal compile-time analysis can determine array access patterns by describing the index variable as a chain of recurrences [2]. For example, the induction variable `i` in loop 2a is incremented by one at each iteration and forms the recurrence $i = i + 1$. Therefore the

access stride is one and loads from this index can be vectorized. Because `j` is conditionally updated, unlike `i`, it cannot be described as a recurrence in general. However, for loop iterations `i=0` and `i=1` corresponding to the uniform-false region of the input data in 2d, `j` does form a recurrence, $j = j + 1$. This insight allows consecutive memory access patterns to be detected and vectorized, such as the store on array `d`.

Probability-Based Cost Model to Determine the Profitability of Run-time Techniques. It is unknown at compile-time how often uniform paths will actually be taken, therefore prior works rely on run-time tuning or profiling to determine the profitability of run-time techniques. The auto-tuning cost model proposed by Sujon et al. [31] is taken as one representative example to illustrate common issues existing in prior work. This auto-tuning cost model considers all combinations of control-paths without dependence (such as the if-branch in Figure 2a) and compares the execution times of vectorizing each candidate control-path. The best-performing path is selected and compared against the performance of scalar code. This cost model is not a feasible solution for automatic compilers due to its expensive tuning overhead. Motivated by issues in prior work, this work proposes one probability-based cost model, which uses branch probability information from a program's execution profile to predict the profitability of vectorization. First, the set of all branch conditions B is found such that B does not contain any statically uniform conditions (for example, no loop invariant conditions). Next, for each branch condition, the cost of applying the VecRC transformation is calculated based on the branch probability. If the sum of these costs is less than the estimated cost of a scalar loop, then the VecRC run-time check technique is considered profitable. Otherwise, the loop is not transformed. See Section 4 for a detailed explanation of VecRC's cost model. Unlike an auto-tuning cost model, the VecRC cost model only requires that each loop be executed once. Additionally, branch probability information is a standard feature offered by all widely-used compilers (such as GCC, ICC, and Clang) and does not require any special implementation, unlike other profiling-based cost models proposed by prior work [3, 26, 28].

3 VECRC: COMPILE-TIME ANALYSIS BASED ON DYNAMIC UNIFORMITY

VecRC performs uniformity-based dependence analysis and scalar evolution analysis to improve the vectorization of control-paths with control-dependent loop-carried dependences. In this section, we describe the structure of the VecRC run-time check, as well as the basic idea behind leveraging run-time uniformity information for compile-time analysis ahead of time.

3.1 Dependence Analysis Based on Dynamic Uniformity

VecRC's dependence analysis supports control-dependent loop-carried dependences by taking advantage of dynamic uniformity. There are three cases of register or memory dependence (shown in Figure 3) that can be supported through the property of uniformity; either by scalarization (Figure 3d and 3e), or through vectorization if certain dependences are no longer satisfied (Figure 3f). A data-dependence graph (DDG) is a directed graph with nodes to represent program components (such as memory operations or variable assignments), and edges to represent dependence relationships between nodes [8, 13]. Using the data-dependence graph implemented in LLVM [15], VecRC determines the category of loop-carried dependence. Such dependences occur as strongly connected components (SCC) within the DDG that prevent vectorization [8].

Intra-Predicated Dependence. Figure 3a shows an example of intra-predicated dependence, where the SCC occurs only within a predicated block (Figure 3d). In the uniform-true case, instructions involved in the SCC and in the control-dependent block are guaranteed to be executed exactly once for every SIMD lane, and can be directly scalarized (shown Figure 3g). Any instructions in the uniform paths that are not a part of the SCC can be vectorized without predication, because they are data-parallel and in a block with a uniform predicate (as shown in the motivating example, Figure 2c).

Partially-Predicated Dependence. Figure 3b shows an example of partially-predicated dependence, where the SCC includes both a control-dependent block t, and the block s it is control-dependent on, $s \in cblocks(t)$. In this case, the SCC contains a mixture of control-dependent and non-control-dependent instructions, therefore the control-dependent instructions cannot directly be scalarized as in the intra-predicated case. Instead, control-dependent instructions are unrolled such that each instance is guarded by the branch instruction they depend on, and control-dependence is not violated (shown Figure 3h). Because this branch instruction must also define the block predicate, each unrolled branch instance also computes the block predicate value for the corresponding lane. Next, VecRC inserts run-time checks to test whether the block predicate is uniform-true (all lanes are active) or uniform-false (all lanes are inactive). Any instructions not involved in the dependence cycle can be directly vectorized without predication, for example, the stores and loads on arrays `a` and `b`.

Inter-Predicated Dependence. Figure 3c shows an example of the inter-predicated dependence case, where dependence edges can be removed from the DDG by uniform predicates. In this example, the dependence has a distance of 1, indicating a SCC that directly disables if-conversion. However, each control-dependent block is mutually exclusive (one or the other will execute for each iteration, but not both). Therefore, all blocks along the uniform paths are also mutually exclusive. Any reads or writes that occur in different paths cannot affect instructions between iterations of the SIMD width, therefore instructions in the uniform branches no longer form an SCC and can be vectorized. However, instructions in the divergent path (shown Figure 3i) are still contained in an SCC and must be scalarized.

3.2 Scalar Evolution Analysis Based on Dynamic Uniformity

Typical scalar evolution analysis, such as implemented in GCC and LLVM, aims to describe scalar values as a chain of recurrences [2] that describe how a value changes over loop iterations [33]. For example, an induction variable i starting from 0 and having stride 1 can be represented as the scalar evolution expression `{0, +, 1}` to indicate i increases by 1 at each iteration (also called an add recurrence). Stride information of induction variables is helpful

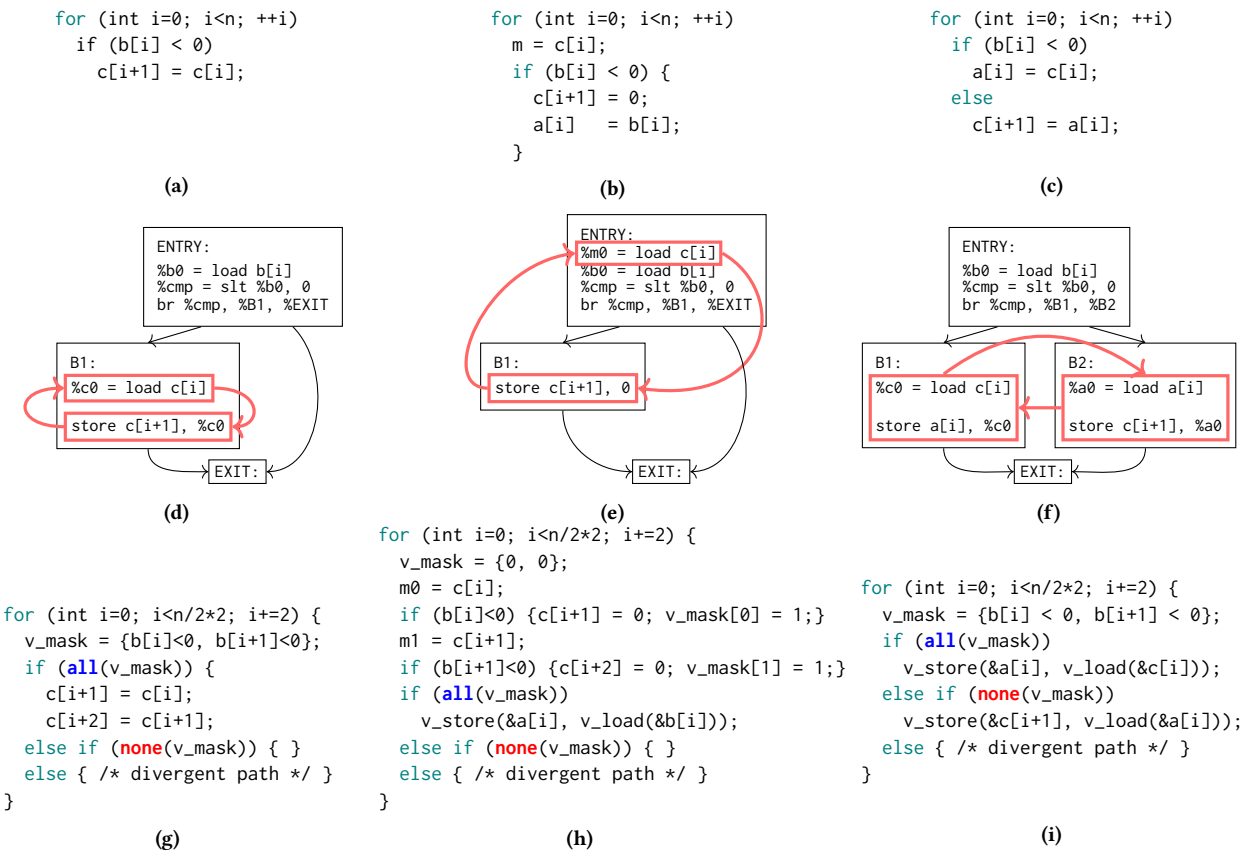


Figure 3: Examples of the three types of control-dependent loop-carried dependences supported by VecRC. (3a) Loop exhibiting an intra-predicated dependence. (3b) Loop exhibiting a partially-predicated dependence. (3c) Loop exhibiting an inter-predicated dependence. (3d), (3e), (3f) are the CFGs of each loop body. (3g), (3h), (3i), are the loops after applying the VecRC run-time check technique to each loop.

for loop auto-vectorization because loads or stores can be directly vectorized if their access function is known to have a stride of 1 at compile-time. In general, the stride of control-dependent scalars cannot be analyzed at compile-time if the control flow depends on input data (such as the motivating example in Figure 2a).

VecRC performs extra scalar evolution analysis based on dynamic uniformity. The fundamental issue with forming chains of recurrences to analyze stride for control-dependent scalars is that, from any iteration to the next, the stride may change. Thus, stride information is linked to control-dependence. If control flow information becomes known, in this case through run-time checks for uniformity, then it can be used at compile-time to determine exact stride information that is safe for vectorization. In the motivating example loop 2a with input data 2d, the variable `j` only has a stride 1 between iterations 0 and 1 (the uniform-false region). Along the uniform-true and uniform-false paths, the value of the block predicate is known at compile-time. VecRC attempts to form a chain of recurrences to find potential induction variables along each uniform path. If the block predicate is all false (uniform-false path), then `j` has the behaviour `{j, +, 1}`, and the stride is exactly 1. Any memory accesses that use `j` as an access function can be vectorized in the uniform-false path. Without performing this analysis, such memory accesses must be predicated.

4 PROBABILITY-DRIVEN COST MODEL

VecRC proposes a probability-based cost model to predict the most profitable strategy for auto-vectorization. Many factors must be considered when estimating profitability, such as the overhead from run-time checks, benefit from the uniform branches, and the cost of the divergent path when a condition is not uniform. VecRC uses general profiling information, available from all widely-used compilers, to consider the impact of these factors and predict profitability.

Profile-Guided Optimization (PGO, also called feedback-directed optimizations or FDO) is a general technique that uses a program's execution profile to either expose new optimization opportunities not detected by static analysis, or improve cost-benefit analysis [10]. The exact types of profile information collected by different compilers may vary, but all widely used compilers such as GCC, Clang, and ICC collect branch probabilities. Based on a branch instruction b with probability p, which is known from PGO information, each uniform and divergent path has a different probability of being

Algorithm 1 VecRC's Probability-Based Cost Model

Input: VF: The set of vectorization factors > 1
P: The probability table, which contains the probability of a branch br condition being true
R: The set of all branches with dynamically uniform conditions

Output: $rt\_check\_decisions$: a structure that stores the mapping $\langle br, vf \rangle \mapsto \langle cost, decision \rangle$
$divergent\_decisions$: a structure that stores the mapping $\langle br, vf \rangle \mapsto \langle cost, decision \rangle$

```
1:  for all v ∈ VF do
2:      for all branch instructions b ∈ R do
3:          // Phase 1: find the minimum cost decision
            // for the divergent path
4:          p := P[b]
5:          T, F := GET_TRUEFALSE_PATHS(b)
6:          cost_SCA := SCALARIZE_COST(p, v, T, F)
7:          cost_IF := IFCONVERT_COST(v, T ∪ F)
8:          if cost_IF < cost_SCA then
9:              divergent_decisions[⟨b, v⟩] := ⟨cost_IF, if-convert⟩
10:         else
11:             divergent_decisions[⟨b, v⟩] := ⟨cost_SCA, scalarize⟩
12:         end if
13:         // Phase 2: determine profitability of
            // the run-time check technique
14:         UT, UF := GET_UNIFORM_PATHS(b)
15:         cost_RT :=
              RT_COST(p, v, UT, UF, divergent_decisions[⟨b, v⟩].first)
16:         if cost_RT < divergent_decisions[⟨b, v⟩].first then
17:             rt_check_decisions[⟨b, v⟩] := ⟨cost_RT, true⟩
18:         else
19:             rt_check_decisions[⟨b, v⟩] :=
                    ⟨divergent_decisions[⟨b, v⟩].first, false⟩
20:         end if
21:     end for
22: end for
23: // Phase 3: choose the minimum cost vectorization factor
24: return SELECT_VECTORIZATION_FACTOR(rt_check_decisions)
```

executed. We assume that the branch probability p in a specific iteration is independent from that of other iterations. Thus, loops with a loop-carried dependence in their branch condition are not supported in our cost model. Under this assumption, the probability of executing the uniform-true path is p^v, where v is the vectorization factor (number of SIMD lanes). Symmetrically, the probability of taking the uniform-false path is $(1-p)^v$. The probability of taking the divergent branch is then $1 - p^v - (1-p)^v$. These three probabilities correspond to the likelihood of each uniform-true path UT, uniform-false path UF, or divergent path DV being executed at run-time. In the following sections, we present how VecRC's cost model (shown in Algorithm 1) incorporates this compile-time prediction.

Phase 1: Decision for Divergent Path. During phase 1, the cost model makes a decision for the divergent path that is a key-value mapping between a branch/vectorization factor pair (the key) and a vectorization decision/associated cost pair (the value) as shown in lines 9 and 11. If the divergent path does not contain any control-dependent loop-carried dependence, then it can either be executed as vector code (through if-conversion) or scalar code. In some cases, scalarization may be more profitable than vectorization. For example, if the divergent path is executed with low probability, vectorization using masked instructions will consequently have low lane utilization; scalar code may achieve better performance. Thus, branch probability is an important factor when deciding whether to vectorize or scalarize the divergent path.

Each candidate branch instruction $b \in R$ has a set of blocks $T \cup F$ (line 5) that are control-dependent on the block terminated by b. The condition of branch instruction b will evaluate to true with probability p (line 4). Additionally, each block contains instructions with an associated cost. Let i be a scalar instruction inside a block $B \in T \cup F$, then $\text{seq}(i)$ is the cost of instruction i. The function $\text{ifconv}(v, i)$ is the cost for the vector instruction sequence of scalar instruction i. The cost of a block B is the sum $\sum_{i \in B} \text{fn}(i)$ of all instruction costs in B (where fn is seq or ifconv). The cost model calculates the cost of either scalarization or vectorization of the divergent path using the following equations:

$$\begin{aligned} \text{SCALARIZE\_COST}(p, v, T, F) = &\sum_{B \in T} \sum_{i \in B} \text{seq}(i) \cdot v \cdot p \\ &+ \sum_{B \in F} \sum_{i \in B} \text{seq}(i) \cdot v \cdot (1-p) \end{aligned} \tag{1}$$

$$\text{IFCONVERT\_COST}(v, B_{\text{PD}}) = \sum_{B \in B_{\text{PD}}} \sum_{i \in B} \text{ifconv}(i) \tag{2}$$

As the branch probability p varies, so does the number of scalar instructions that will be executed in the divergent path (Equation 1). If the divergent path is vectorized, each instruction will always be executed due to if-conversion and therefore the cost does not depend on p (Equation 2). On line 8, the cost model uses equations 1 and 2 to compare the costs of either scalarizing or vectorizing the divergent path. The decision with lowest cost is stored in $divergent\_decisions$ as a mapping between branch and vectorization factor to associated cost and decision (lines 9 and 11). If the divergent path contains a dependence, if-conversion is an illegal transformation and the associated cost is equal to infinity. This cost/decision pair is later used in phase 2 to calculate the overall cost of applying the run-time check technique.

Phase 2: Decision for Run-time Checks. The cost of the run-time check technique is divided into four components: uniform-true path cost, uniform-false path cost, divergent path cost, and the cost of detecting uniformity (run-time check overhead). Additionally, the uniform and divergent path costs depend on the branch probability p. For example, if the divergent path is executed frequently, its cost may dominate and cause the run-time check technique to be unprofitable. The cost of a path $s \rightarrow t$ is the sum of all block costs $b \in s \rightarrow t$ and is defined as $cost(s \rightarrow t)$. Additionally, let RT be the set of instructions that implement the run-time checks. These instructions are always executed (with probability $p = 1$). Then, the total expected value of the entire VecRC transformation is calculated as:

$$\begin{aligned} cost_{\text{DV}} &= divergent\_decisions[\langle b, v \rangle].first \\ \text{RT\_COST}(p, v, UT, UF, cost_{\text{DV}}) &= \sum_{i \in RT} \text{seq}(i) + cost(UT) \cdot p^v \\ &\quad + cost(UF) \cdot (1-p)^v \\ &\quad + cost_{\text{DV}} \cdot (1 - p^v - (1-p)^v) \end{aligned} \tag{3}$$

On line 15, the cost model compares the total cost of the run-time check technique to the minimum divergent path cost calculated in phase 1. The run-time check technique is profitable if its cost is less than the best decision for the divergent path. Otherwise, the run-time check technique will not be applied for the current branch instruction and vectorization factor; instead, the decision from phase 1 is most profitable. This profitability decision is stored in the *rt_check_decisions* structure as a mapping between branch instruction and vectorization factor to associated cost and run-time check technique disabled/enabled (represented as a boolean value). Thus, phase 2 computes the most profitable decision for all branch instruction/vectorization factor pairs.

Phase 3: Select Vectorization Factor. After phases 1 and 2 are complete, the optimal decisions and associated costs for each branch instruction/vectorization factor pair are stored in the structures *rt_check_decisions* and *divergent_decisions*. Phase 3 selects the vectorization factor with lowest cost for all branch instructions in the loop body:

$$v_{\text{opt}} = \underset{v \in VF}{\arg\min} \left(\sum_{b \in B} rt\_check\_decisions[\langle b, v \rangle].first \right) \quad (4)$$

If the cost associated with v_{opt} is less than the cost of the scalar loop, then the run-time check technique is enabled. Once the loop is being transformed, decisions are recovered through the mappings defined in earlier phases; for example, the decision whether to enable run-time checks for branch instruction b with vectorization factor v_{opt} is stored as $rt\_check\_decisions[\langle b, v_{\text{opt}} \rangle].second$. The cost model can make individual decisions for each branch instruction within the loop body, in other words, the decision to enable or disable the run-time check technique is not global among all branch instructions.

The main benefits of this cost model are that it 1) does not require special profiling or expensive auto-tuning; 2) is parameterized by vectorization factor, unlike prior work that must collect profiling for each factor [28]; 3) relies on instruction cost information offered by any widely-used compiler, and therefore can be applied to a compiler-supported target architecture without modification.

Implementation. VecRC determines which vectorization factor to use by selecting a vectorization plan (VPlan) with the lowest associated cost. A VPlan is a model within LLVM for describing different vectorization transformations, without modifying the intermediate representation [24]. VecRC's VPlan consists of three paths, one for each of the uniform-true, uniform-false, and divergent paths. In phase 1 of Algorithm 1, VecRC's VPlan uses decisions from the cost model to determine whether if-conversion or scalarization is profitable for a divergent path. To decide if the VPlan should include a run-time check, the cost model is used to calculate run-time check profitability (phase 2 of Algorithm 1). After phase 1 and 2, for every vectorization factor, one specific VPlan is constructed. During phase 3, the VPlan/vectorization factor with the lowest associated cost is selected. VecRC's cost model uses the BranchProbabilityInfo analysis provided by LLVM to obtain branch weights from profile information [23]. If profile information is not available, branch weights are estimated using heuristics implemented in the BranchProbabilityInfo analysis.

5 EVALUATION

VecRC is implemented in LLVM (13.0.0) and based on the existing loop-level vectorizer. We modify the legality analysis, cost analysis, and code generation modules to realize the VecRC compiler.

5.1 Methodology and Benchmarks

Architectures. We evaluate VecRC on the Intel Skylake and Power 9 architectures shown in Table 2. Both processors support vector extensions with different SIMD widths and different instruction support. For example, AVX512 supports predication through masking, but VSX does not.

Prior Work. We compare VecRC against Clang and Region Vectorizer (which implements compile-time and run-time techniques based on uniformity [20, 28]). Despite our best efforts to compare against one additional prior work based on Region Vectorizer, WCCV [32], we were unable to successfully compile the selected benchmarks.

Benchmarks. We select eight benchmarks to evaluate our VecRC implementation. The benchmarks are selected based on two metrics: 1) they have the code patterns VecRC supports; 2) they are widely used and cited in prior work [3, 12, 16, 19, 32, 34]. The TSVC-2 benchmark suite [18] is used to compare VecRC's vectorization rate against other compilers, as well as evaluate the accuracy of the probability-based cost model.

Setup. To ensure a fair comparison against Clang and Region Vectorizer, the same LLVM-IR is used as input to every vectorization pass implemented by each compiler. Additionally, because Region Vectorizer does not perform legality analysis and requires manual annotation from the user to enable vectorization, we annotate one bottleneck loop in each benchmark to indicate that it is safe to vectorize for Region Vectorizer. For experiments that compare against Region Vectorizer, all compilers vectorize the single bottleneck loop only. To generate the input LLVM-IR, an unoptimized binary is compiled by Clang with PGO enabled to collect the necessary profiling information. Next, that profiling information is used to generate unoptimized LLVM-IR from the file containing the target loop; profiling information is encoded as metadata in the resulting IR. The unoptimized IR with PGO information is used as input to the `opt` tool, which applies the `mem2reg`, `loop-simplify`, and `simplify-cfg` passes. The output from `opt` is used as input for each vectorization pass from Clang (LLVM's Loop Vectorizer), Region Vectorizer, and VecRC. The resulting IR is linked with other required objects (compiled using the same optimization pipeline) to generate a binary. Because Region Vectorizer has a different optimization pipeline than Clang and VecRC, this strategy guarantees a fair comparison between different loop vectorization passes. However, since Clang and VecRC share the same optimization pipeline, we also evaluate the performance benefit from different loop vectorization passes using the O3 pipeline. We use the single-thread performance reported by each benchmark; the run-time check overhead is included implicitly.

5.2 TSVC-2 Loops

Comparison of Vectorization Rates. The TSVC-2 benchmark contains 34 loops with 1) control-dependent predicated blocks, and 2) dynamically uniform branch conditions. On both Skylake

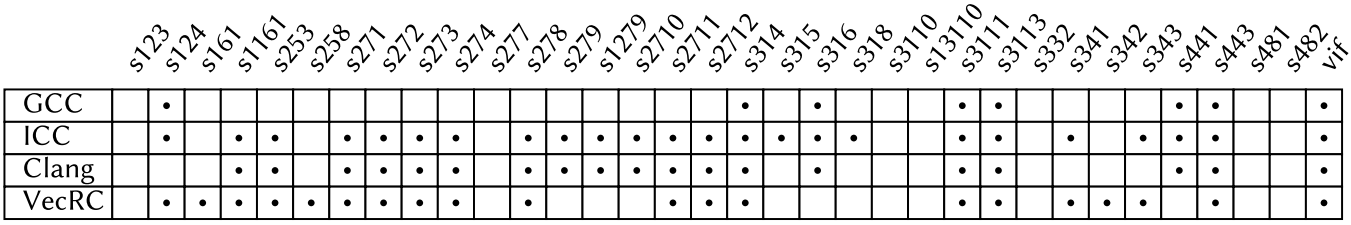


Figure 4: 34 loops from TSVC-2. A dot shows that the corresponding compiler can vectorize the inner loop of the listed function.

Name	Domain	Suite
519.lbm_r	Computational fluid dynamics	SPEC2017
525.x264_r	Video compression	SPEC2017
Moldyn	Particle Simulation	
EP	Random-number generation	NPB
Cutcp	Cutoff Pair Potential for Molecular Modeling Applications	Parboil
Mri-Gridding	Magnetic Resonance Imaging	Parboil
CFD	Finite volume solver for three-dimensional Euler equations	Rodinia
miniMD	Simulates the Newtownian equations of particle motion	

Table 1: Selected Benchmarks

Architecture	CPU	Vector Extension
Skylake	Xeon W-2145 (3.70GHz)	AVX512 (512-bit)
Power ISA	Power 9	VSX (128-bit)

Table 2: Evaluation Architectures

and Power 9 architectures, GCC, ICC (Skylake only), Clang, and VecRC vectorize 8, 24, 19, and 20 of the total 34 loops respectively (shown Figure 4). Because Clang and VecRC are both based on LLVM, VecRC vectorizes one additional loop over Clang due to uniformity-based dependence analysis. Clang vectorizes five loops, `s279`, `s1279`, `s2710`, `s316`, and `s441` that VecRC cannot. This is because the loops (except `s316`) have complex control-flow such as nested if-statements, which is not currently supported in our compiler. ICC recognizes three cases of reductions in loops `s315`, `s316`, and `s318` that VecRC does not vectorize. This is because VecRC's cost model does not support min/max style reductions that contain loop-carried dependences in the branch condition. VecRC vectorizes the three loops `s161`, `s258`, and `s342` that ICC cannot due to control-dependent loop-carried dependences.

5.3 Real-world Benchmarks

Performance as a Single Vectorization Pass. On the Skylake architecture, VecRC attains a geometric mean speedup over a scalar baseline of 1.32x on 6 real-world benchmarks (Figure 5a) compared to 1.11x and 1.00x for Clang and Region Vectorizer, respectively. On the Power 9 processor, VecRC achieves a geometric mean speedup of 1.22x over 1.01x and 1.00x for Clang and Region Vectorizer, respectively (Figure 5b). Figures 5a and 5b show the performance benefits from run-time techniques implemented in Region Vectorizer are insignificant on both Skylake and Power 9 architectures.

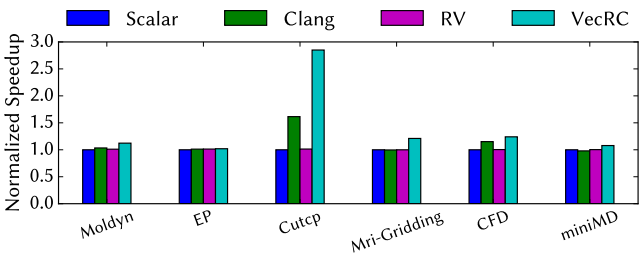
The results show that run-time techniques combined with the compile-time analysis implemented in VecRC can outperform the if-conversion technique used by mainstream compilers, such as Clang. When data-dependence does not exist and dynamic uniformity occurs (Cutcp, CFD), the benefits from run-time checks amortize run-time check overhead and outperform the if-conversion method implemented in Clang. When data-dependence exists (Moldyn, EP, Mri-Gridding and miniMD) and dynamic uniformity occurs, Clang and Region-Vectorizer disable loop vectorization; however, VecRC can still partially vectorize such loops through additional dependence analysis based on dynamic uniformity. Region Vectorizer doesn't achieve significant speedup due to two main reasons: 1) run-time checks are often disabled due to an overly conservative cost model; 2) the proposed approach focuses only on control dependence and doesn't support data dependence.

In the highest speedup case of VecRC over scalar code on Skylake, 2.8x (*cutcp*), the predicated block contains only one store operation, has a high compute-to-memory ratio, and does not contain any loop-carried dependences. Therefore, both Clang and VecRC achieve good speedup over scalar code. VecRC achieves speedup over Clang by reducing predication overhead and skipping redundant computation. Without masked instruction support on Power 9, vectorization through if-conversion has a high predication overhead, and Clang is unable to achieve speedup over scalar code.

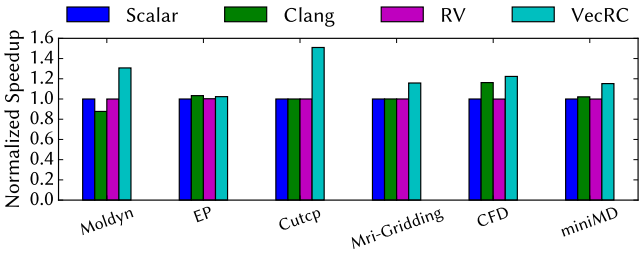
VecRC is most effective when dynamic uniformity is abundant, because the uniform paths are more likely to be executed. VecRC also often provides good speedup when a loop contains control-dependent loop-carried dependence (for example, the loop types in Figure 3) because unlike other compilers VecRC can vectorize such loops using uniformity-based dependence analysis (Section 3.1).

Performance as a Pass in the O3 Pipeline. This experiment (shown Figure 6) aims to evaluate the performance of VecRC as a loop vectorization pass in Clang's O3 pipeline. The auto-vectorization pass is applied to all the source code files in the selected benchmarks (automatically, without any manual annotations).

VecRC+O3 achieves a geometric mean speedup of 1.18x and 1.23x over Clang+O3 on Skylake and Power 9, respectively. To show the benefit from vectorization, we also compare the normalized speedup when vectorization is disabled, VecRC+O3 (No-Vec). On Power 9, Clang+O3 and VecRC+O3 (No-Vec) perform similarly across benchmarks (Figure 6b), indicating that the benefit from vectorization through if-conversion is not high. Power 9 has a

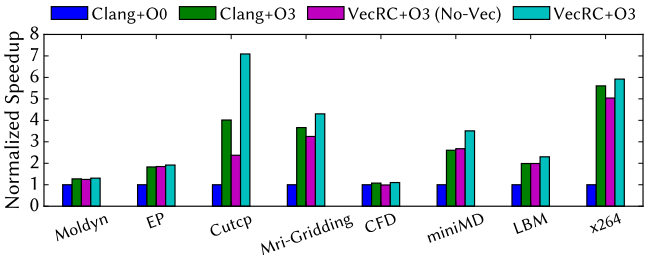


(a) Performance Comparison on Intel Skylake

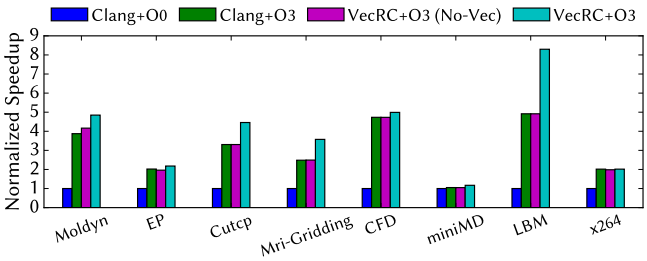


(b) Performance Comparison on Power 9

Figure 5: The performance of Loop-Vectorization Pass in VecRC compared to Clang and Region Vectorizer



(a) Performance Comparison (O3) on Intel Skylake



(b) Performance Comparison (O3) on IBM Power 9

Figure 6: The Performance of VecRC+O3 compared to Clang+O3 on Skylake and Power 9. All execution times are normalized to a scalar baseline (Clang+O0).

lower register width (resulting in a lower maximum vectorization factor) and does not support masking; a lower vectorization factor increases the probability of dynamic uniformity, and architectures that do not support masking benefit from vectorized loads and stores. Combined, these factors allow VecRC to achieve speedup. Additionally, although the *x264* benchmark contains no bottleneck loops, VecRC achieves a 1.06x speedup over Clang+O3 on Skylake; this is an example of the accumulated benefit from applying run-time techniques on many small loops.

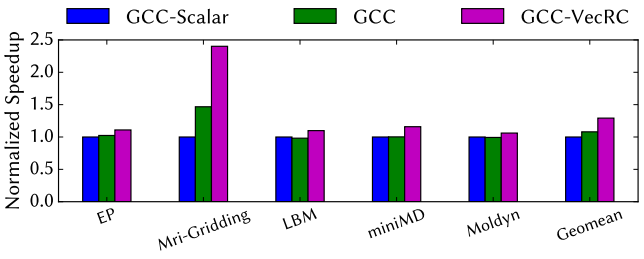
Comparison With GCC and ICC. We are unable to reimplement the VecRC approach inside GCC and ICC, due to the expensive engineering effort (and because the ICC source code is not public). To guarantee a fair comparison with GCC/ICC, we manually implement the VecRC run-time check technique at the source code level to compare against the loop vectorizers of GCC and ICC on Skylake. Figure 7 shows the performance comparison between VecRC, GCC, and ICC on Intel Skylake. While GCC's legality analysis tends to be conservative, ICC is optimized to take advantage of masked instructions supported by AVX512. Although the manually implemented version cannot benefit from cost model decisions (such as vectorization factors), the manual VecRC implementations compiled by GCC and ICC attain geometric mean speedups of 1.20x and 1.06x respectively over the compiler auto-vectorization on 5 real-world benchmarks (Figure 7).

5.4 Performance Breakdown

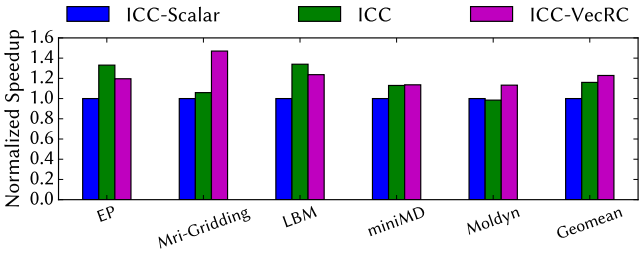
In Figure 8, we separate the effect of dependence analysis and scalar evolution analysis based on dynamic uniformity. All five benchmarks contain a control-dependent loop-carried dependence that Clang fails to vectorize. When VecRC's dependence analysis is disabled (RTC+NoDep), the execution time matches Clang. The s124, s341, and s342 benchmarks additionally contain consecutive memory access patterns that can only be vectorized using VecRC's scalar evolution analysis. Therefore, when vectorization is enabled but scalar evolution is not (RTC+Dep), these benchmarks experience a slowdown due to scalarized memory operations (normally, the cost model would disable run-time checks). Once scalar evolution analysis is enabled (RTC+Dep+SCEV), speedup is achieved over RTC+Dep for s124. For the s341 and s342 loops, the benefit from vectorization is trivial, and neither RTC Disabled nor VecRC can achieve speedup. In these cases, VecRC will disable vectorization and match the performance of scalar code. VecRC achieves speedup for the EP and Mri-Gridding benchmarks due to the partial vectorization enabled by dependence-analysis based on dynamic uniformity.

5.5 Compilation Overhead

VecRC introduces compile-time overhead due to the additional dependence analysis, scalar evolution analysis, and cost analysis that is not performed by LLVM's loop vectorizer. Table 3 compares the compilation times of Clang, No-Vec (where vectorization in the VecRC compiler is disabled), and VecRC for eight benchmarks on the Skylake architecture. VecRC's run-time check technique introduces a compile-time overhead of approximately 13%. In smaller benchmarks such as LBM, VecRC's compile-time overhead is more apparent, taking approximately 30% longer to compile than Clang. For the longest benchmark to compile, x264, loop vectorization



(a) Comparison between GCC auto-vectorization and manual VecRC vectorization. Speedups are normalized to GCC scalar execution time.



(b) Comparison between ICC auto-vectorization and manual VecRC vectorization. Speedups are normalized to ICC scalar execution time.

Figure 7: The performance of VecRC implemented at the source-code level compared to auto-vectorization in (7a) GCC and (7b) ICC on the Skylake architecture.

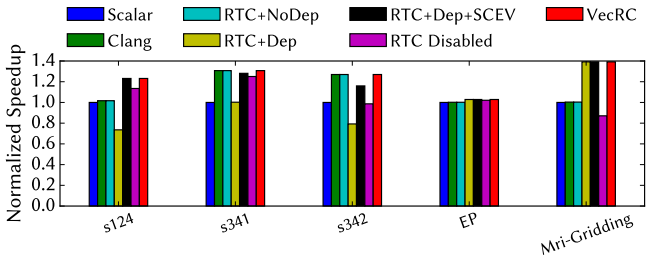


Figure 8: The benefit from each compile-time analysis. RTC stands for run-time check. RTC+NoDep: RTC is enabled, dependence analysis is disabled. RTC+Dep: RTC is enabled, dependence analysis is enabled. RTC+Dep+SCEV: runtime-check, dependence analysis, and scalar evolution analysis are enabled. All execution times are normalized to the scalar baseline.

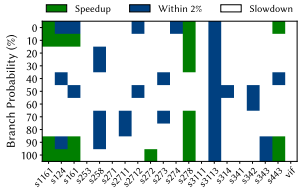
does not dominate compile time, therefore both VecRC and Clang compile the benchmark in approximately the same amount of time.

5.6 Cost Model Accuracy

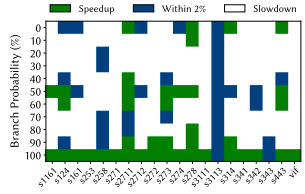
The probability-based cost model was evaluated across the 20 TSVC-2 loops VecRC can vectorize and 11 different branch probabilities (each branch probability is obtained by modifying the loop input data randomly). The TSVC-2 benchmark was selected specifically

	Clang (s)	No-Vec (s)	VecRC (s)	Ratio (%)
Moldyn	0.15	0.13	0.16	6.6
EP	0.19	0.20	0.21	10.5
Cutcp	0.37	0.37	0.43	16.2
Mri-Gridding	0.25	0.24	0.30	20
CFD	0.30	0.29	0.32	6.6
miniMD	1.55	1.57	1.84	18.7
lbm	0.46	0.46	0.61	32.6
x264	23.42	22.73	22.85	-2.4
			Geomean (%)	13.1

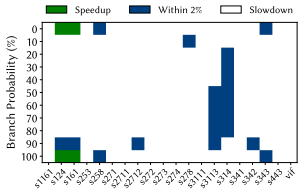
Table 3: Comparison of O3 compilation time on Intel Skylake (in seconds).



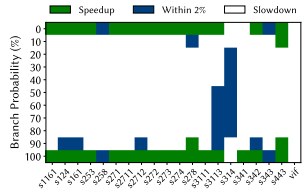
(a) Cost model predictions, vectorization factor 2.



(b) Real performance, vectorization factor 2.



(c) Cost model predictions, vectorization factor 4.



(d) Real performance, vectorization factor 4.

Figure 9: Comparison of cost model predictions and real profitability. Blue: The baseline and VecRC transformation have the same running time within 2%. Green: the cost model predicted speedup, or the real execution time of VecRC was faster than the baseline. White: the cost model predicted slowdown, or the real execution time of VecRC was slower than the baseline.

for this experiment because the loops contain very few instructions in the uniform paths, and therefore the threshold between profitability and non-profitability is small. To determine whether the cost model correctly predicts run-time check profitability for a given loop, branch probability, and vectorization factor, the result from cost analysis is compared against real performance (Figure 9). Real performance is a comparison between two loop versions: run-time check disabled and run-time check enabled. In the run-time check disabled version, all unpredicated blocks are vectorized, and any predicated blocks are scalarized or vectorized through if-conversion (whichever is lowest cost). In the run-time check enabled version, the run-time check technique is applied to all candidate branches, thus creating vectorized versions of predicated

blocks. The run-time check disabled version is used as the baseline, instead of scalar loops, to ensure that any benefit from vectorized unpredicated blocks is not a contributing factor to real speedup with the run-time check enabled version.

A *false positive* occurs when the cost model predicts speedup, but the run-time check technique results in a slowdown. Conversely, a *false negative* occurs when the cost model predicts a slowdown, but a speedup was observed when applying the run-time check technique and executing the loop. If the cost model predicted either a speedup or slowdown, but real execution time was within 2%, we consider this neither a false positive nor a false negative. Figure 9 shows a comparison between predicted profitability (left-hand side) and real performance (right-hand side) of different vectorization factors. For vectorization factors 2 and 4, the cost model has a false negative rate of 13.6% and 23.2%, respectively. As the vectorization factor increases, the likelihood of executing uniform paths decreases, and the cost model becomes more conservative. As a result, the false positive rate decreases as the vectorization factor increases: our cost model has a false positive rate of 4.5% and 0% for vectorization factors 2 and 4, respectively. Through these experiments (of 440 test cases in total), we demonstrate that the probability-based cost model can correctly predict run-time check profitability in practice.

6 RELATED WORK

Modern compilers employ two main techniques for auto-vectorization: 1) loop auto-vectorization merges multiple scalar operations across consecutive loop iterations into a single SIMD instruction [29], 2) Superword Level Parallelism (SLP) merges isomorphic instructions in the same block into a single SIMD instruction [14].

The main strategy for vectorizing control flow is if-conversion [1], which converts control flow dependence to data flow dependence thus enabling vectorization through predication. Shin et al. propose a technique to support control flow in the SLP method by using `select` instructions to merge control paths [27]. Similarly, Pohl et al. combine `select` instructions with a run-time check to enable loop auto-vectorization of control flow without masked instructions [21]. Previous work has proposed several techniques to auto-vectorize loops with control flow and control-dependent loop-carried dependences [3, 31].

Karrenberg et al. [11] propose a compile-time approach that avoids if-converting uniform branches, which Moll et al. [20] improve to avoid creating irreducible loops and decrease algorithmic complexity. There are two main run-time approaches to avoid if-converting uniform branches. The Branch-On-Superword-Condition-Codes (BOSCC) strategy [27, 28] inserts run-time checks to detect uniform-false block predicates and skip redundant computation. WCCV [32] proposes a run-time check to detect uniform-true predicates and avoid predication overhead (for example, by eliding masked instructions). To improve the effectiveness of such run-time techniques on architectures with very wide registers, existing work [22] proposes a strategy to construct dynamic uniformity at run-time by grouping active and inactive lanes into separate registers. While this method can increase dynamic uniformity, it introduces additional overhead and is limited to data-parallel loops.

Strategies are proposed to overcome the limitations of static analysis and execute more aggressive optimizations based on information from run-time checks [7, 25]. These works follow a common approach of constructing a run-time check at compile-time that verifies certain constraints of conservative static analysis, for example dependence information. The run-time check determines if an optimized or fallback loop version should be executed, based on whether the optimization is valid at run-time. A drawback of this approach is that different run-time checks are required for different analyses.

7 FUTURE WORK AND CONCLUSION

In this paper, we demonstrated that combining run-time checks for dynamic uniformity with compile-time analysis improves control-flow vectorization. Our proposed technique, VecRC, takes a loop, dynamic uniformity information from run-time checks, and profiling information as inputs to apply uniformity-based analyses at compile-time. VecRC's cost model determines the most profitable decision for control-flow vectorization. By taking advantage of dynamic uniformity at both compile-time and run-time, VecRC outperforms existing uniformity-based vectorization techniques and enables vectorization even for loops with control-dependent loop-carried dependences.

VecRC can be extended to support a wide range of irregular loops with features that hinder or disable traditional loop vectorization. In future work, we plan to support more complex control-flow, such as nested if-statements. Additionally, we plan to adapt VecRC to *vector-length agnostic* architectures (such as ARM SVE [30]) and architectures with very wide vector registers. Along this direction, we plan to evaluate approaches in which blocks are vectorized using different vectorization factors (VF), where unpredicated blocks are vectorized with a high VF while predicated blocks are still vectorized using our current approach but with a lower VF to take advantage of dynamic uniformity.

REFERENCES

[1] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Austin, Texas) *(POPL '83)*. Association for Computing Machinery, New York, NY, USA, 177–189. https://doi.org/10.1145/567067.567085

[2] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. 1994. Chains of Recurrences—a Method to Expedite the Evaluation of Closed-Form Functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation* (Oxford, United Kingdom) *(ISSAC '94)*. Association for Computing Machinery, New York, NY, USA, 242–249. https://doi.org/10.1145/190347.190423

[3] Sara S. Baghsorkhi, Nalini Vasudevan, and Youfeng Wu. 2016. FlexVec: auto-vectorization for irregular loops. *ACM SIGPLAN Notices* 51 (6 2016), 697–710. Issue 6. https://doi.org/10.1145/2980983.2908111

[4] Joseph CH Park and Mike Schlansker. 1991. On Predicated Execution. (1991).

[5] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. 2011. Divergence Analysis and Optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 320–329. https://doi.org/10.1109/PACT.2011.63

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[7] Johannes Doerfert, Tobias Grosser, and Sebastian Hack. 2017. Optimistic loop optimization. *CGO 2017 - Proceedings of the 2017 International Symposium on Code Generation and Optimization* (2 2017), 292–304. https://doi.org/10.1109/CGO.2017.7863748

[8] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[9] Agner Fog. 2022. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf

[10] R. Gupta, E. Mehofer, and Y. Zhang. 2002. Profile Guided Compiler Optimizations. In *The Compiler Design Handbook: Optimizations and Machine Code generation*. CRC Press.

[11] Ralf Karrenberg and Sebastian Hack. 2011. Whole-Function Vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 141–150.

[12] Seonggun Kim and Hwansoo Han. 2012. Efficient SIMD code generation for irregular kernels. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 55–64.

[13] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Williamsburg, Virginia) *(POPL '81)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/567532.567555

[14] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[16] Yunsup Lee, Vinod Grover, Ronny Krashinsky, Mark Stephenson, Stephen W Keckler, and Krste Asanovic. 2014. Exploring the design space of SPMD divergence management on data-parallel architectures. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 101–113.

[17] Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, and Krste Asanović. 2013. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–11. https://doi.org/10.1109/CGO.2013.6494995

[18] Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. 2011. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 372–382.

[19] Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. 2019. Revec: program rejuvenation through revectorization. In *Proceedings of the 28th International Conference on Compiler Construction*. 29–41.

[20] Simon Moll and Sebastian Hack. 2018. Partial Control-Flow Linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 543–556. https://doi.org/10.1145/3192366.3192413

[21] Angela Pohl, Biagio Cosenza, and Ben Juurlink. 2018. Control Flow Vectorization for ARM NEON. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems* (Sankt Goar, Germany) *(SCOPES '18)*. Association for Computing Machinery, New York, NY, USA, 66–75. https://doi.org/10.1145/3207719.3207721

[22] Wyatt Praharenka, David Pankratz, João PL De Carvalho, Ehsan Amiri, and José Nelson Amaral. 2022. Vectorizing divergent control flow with active-lane consolidation on long-vector architectures. *The Journal of Supercomputing* 78, 10 (2022), 12553–12588.

[23] LLVM Project. 2022. llvm::BranchProbabilityInfo Class Reference. https://llvm.org/doxygen/classllvm_1_1BranchProbabilityInfo.html.

[24] LLVM Project. 2022. Vectorization Plan. https://llvm.org/docs/Proposals/VectorizationPlan.html.

[25] Diogo N. Sampaio, Louis-Noël Pouchet, and Fabrice Rastello. 2017. Simplification and Runtime Resolution of Data Dependence Constraints for Loop Transformations. In *Proceedings of the International Conference on Supercomputing* (Chicago, Illinois) *(ICS '17)*. Association for Computing Machinery, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3079079.3079098

[26] Jaewook Shin. 2007. Introducing Control Flow into Vectorized Code. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*. IEEE Computer Society, USA, 280–291.

[27] Jaewook Shin, Mary Hall, and Jacqueline Chame. 2005. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society, USA, 165–175. https://doi.org/10.1109/CGO.2005.33

[28] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. 2009. Evaluating Compiler Technology for Control-Flow Optimizations for Multimedia Extension Architectures. *Microprocess. Microsyst.* 33, 4 (jun 2009), 235–243. https://doi.org/10.1016/j.micpro.2009.02.002

[29] N. Sreraman and R. Govindarajan. 2000. A Vectorizing Compiler for Multimedia Extensions. *International Journal of Parallel Programming* 28 (2000), 363–400.

[30] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. 2017. The ARM scalable vector extension. *IEEE micro* 37, 2 (2017), 26–39.

[31] Majedul Haque Sujon, R. Clint Whaley, and Qing Yi. 2013. Vectorization Past Dependent Branches through Speculation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (Edinburgh, Scotland, UK) *(PACT '13)*. IEEE Press, 353–362.

[32] Huihui Sun, Florian Fey, Jie Zhao, and Sergei Gorlatch. 2019. WCCV: Improving the Vectorization of IF-Statements with Warp-Coherent Conditions. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 319–329. https://doi.org/10.1145/3330345.3331059

[33] Robert A. van Engelen. 2001. Efficient Symbolic Analysis for Optimizing Compilers. In *Compiler Construction*, Reinhard Wilhelm (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 118–132.

[34] Hao Zhou and Jingling Xue. 2016. A compiler approach for exploiting partial SIMD parallelism. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 1–26.