A Framework for Fine-Grained Program Versioning

Yishen Chen
MIT CSAIL
ychen306@mit.edu

Saman Amarasinghe
MIT CSAIL
saman@csail.mit.edu

2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 979-8-3503-5057-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/MICRO61859.2024.00024

***Abstract*—Static dependence analysis is critical for optimizations such as vectorization and loop-invariant code motion. However, traditional static dependence analysis is often imprecise, making these optimizations less effective. To address this issue, production compilers use loop versioning to rule out some categories of memory dependencies at run time. However, loop versioning is loop-centric and usually tied to specific optimizations (e.g., loop vectorization), making it less effective for non-loop optimizations such as superword-level parallelism (SLP) vectorization.**

In this paper, we propose a fine-grained versioning framework to rule out program dependencies at run time. Our framework is general and not tailored to any specific optimizations. To use our system, a client optimization specifies groups of instructions (or loops) whose independence is desired but unprovable statically. In response, our system duplicates the appropriate instructions and guards the original ones with run-time checks to guarantee their independence; if the checks fail, the duplicated instructions execute instead.

In a case study, we extended an existing SLP vectorizer with minimal modifications using our framework, resulting in a $1.17\times$ speedup over Clang's vectorizers on TSVC and a $1.51\times$ speedup on PolyBench. In both benchmarks, we encountered programs that could not be vectorized with loop versioning alone.

In a second case study, we used our framework to implement a more aggressive variant of redundant load elimination than the one implemented by Clang. Our redundant load elimination results in a $1.012\times$ speedup on the SPEC 2017 Floating Point benchmarks, with the maximum speedup being $1.064\times$.

I. Introduction

Dependence analysis is crucial for many compiler transformations, such as loop-invariant code motion, vectorization, and, more generally, any optimizations that perform code motion. However, performing precise dependence analysis is difficult. For example, in a language such as C, the compiler must assume that arbitrary pointer arguments may alias and that all memory accesses on those pointers are dependent. This scenario can be addressed, to some degree, with interprocedural pointer analyses. However, such analyses are prohibitively expensive and not deployed in production compilers. Furthermore, dependence analysis is complicated by nontrivial pointer arithmetic and control flow, both of which can induce patterns that are impossible to analyze soundly at compile time, even with state-of-the-art analysis. For example, two indirect pointer accesses could be both dependent and independent, depending on how the input is initialized. All these factors contribute to imprecise dependence analysis, making optimizations less effective as a consequence.

Loop Versioning. When there is insufficient dependence information, production compilers use a technique called *loop versioning*. With loop versioning, the optimizer makes two copies of a loop—one loop is optimized, assuming there are no dependencies among instructions that are critical to the optimization, and the other is left unoptimized—and generates run-time checks so that the optimized loop is only entered if the absence of these dependencies is confirmed. Because the run-time checks must be performed upfront before the loop is executed, loop versioning only works if all the requisite information is known before the loop is executed.

Consequently, loop versioning cannot rule out dependencies that are loop-variant. For instance, complex pointer arithmetic can lead to some memory accesses that are dependent in some iterations yet independent in others. Similarly, loop versioning cannot rule out rare dependencies that are caused by conditionally executed instructions within the loop. To deal with such dependencies in the context of vectorization, researchers have proposed an approach that performs sophisticated run-time checks that are more fine-grained [10, 17]. In this approach, the vectorizer first identifies some specific control-flow paths along which the loop iterations are completely independent. It then optimizes the program assuming only those paths are taken, while also generating the necessary recovery code in case the paths are not taken. Unlike loop versioning, this approach is tied specifically to vectorization and only works in the presence of loops that follow certain patterns.

Our Approach. In this paper, we propose a fine-grained versioning framework to rule out program dependencies at run time. Our framework is general, flexible, and not coupled with specific optimizations. To use our system, a client optimization supplies groups of instructions (or loops) whose independence is desired but cannot be proven statically. In response, our system identifies a small—and in many cases optimal—set of conditions that ensures the independence of those instructions and then versions the input program accordingly.

Our framework is fine-grained because it develops a versioning strategy for each group of instructions (or loops) individually, without having to version whole loops at a time. This is particularly useful when some dependence relationships change from loop iteration to iteration due to nontrivial uses of pointer arithmetic. Furthermore, our framework supports

979-8-3503-5057-9/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00024

nested versioning, which is necessary when one versioning transformation itself relies on assumptions that cannot be statically proven, thus requiring a further round of versioning to confirm those assumptions at run time.

Central to our framework is the concept of *dependence condition*, which identifies the necessary condition under which a dependence occurs at run time. While most dependencies are unconditional—such as when arithmetic instructions depend on their operands unconditionally—others are conditional. For example, a pair of memory accesses is only dependent if the accessed memory locations overlap; a load only depends on a preceding conditional store if the store actually executes at run time. Our insight is that when dealing with instructions whose independence cannot be statically proven due to *conditional* dependencies, we can establish a sufficient condition for their independence by finding a cut over the dependence graph that makes the instructions unreachable from one another. This cut reveals the conditional dependencies that we must check at run time.

To demonstrate the effectiveness of our framework, we first applied our versioning framework to an existing implementation of superword-level parallelism (SLP) vectorization. Although most production loop vectorizers—such as those employed by GCC and Clang—use loop versioning, there has been no implementation of SLP vectorization capable of versioning because it is more flexible and not loop-centric. Consequently, SLP vectorization has been limited by pointer aliasing and other sources program dependencies, particularly when compared to loop vectorization.

We adapted Chen et al.'s [4] SLP vectorizer—an aggressive SLP implementation that can pack independent instructions globally across loops and basic blocks—to use our versioning framework. With versioning, this single implementation of SLP is $1.17\times$ faster than both LLVM's loop and SLP vectorizers combined on the TSVC benchmark (and $1.07\times$ faster than their previous implementation). On PolyBench with the `restrict` keyword disabled, versioning enabled a $1.51\times$ speedup over both LLVM's vectorizers (and $1.62\times$ faster than their previous implementation). In both benchmark suites, we found examples of programs that cannot be vectorized with loop versioning alone. We obtained these results without changing *any* of the vectorizer's heuristic or code-generation algorithms: we only modified the vectorizer to inform our framework of the groups of instructions that should be independent, and it versions the program transparently.

Finally, we used our framework to implement a variant of redundant load elimination that removes redundant loads that cannot be proven redundant with traditional static analysis. With versioning, our implementation of redundant load elimination achieves a geometric mean speedup of 1.2% on the SPEC 2017 Floating Point benchmark suite, with the maximum speedup being 6.4%.

```
Y[0] = 0;
if (*X) cold_func();
Y[1] = 0;
```

Fig. 1: Running example. The highlighted instructions may be dependent and cannot be directly vectorized.

```
noalias = X != Y;
if (noalias) {
  x = *X;
  if (x) Y[0] = 0;
  else {
    Y[0] = 0;
    Y[1] = 0;
  }
} else {
  Y[0] = 0;
  x = *X;
}
if (x) cold_func();
if (!noalias || x) Y[1] = 0;
```

Fig. 2: Running example transformed so that the stores become independent (and amenable to vectorization).

This paper makes the following contributions:

- A general, fine-grained versioning framework.
- The concept of *nested* versioning, where the run-time checks of a versioning transformation require further versioning.
- The concept of *dependence condition*, which allows us to systematically identify dependencies that can be eliminated at run time.
- Empirical evaluations showing that our framework enables an existing SLP vectorizer to effectively vectorize programs in the presence of conditional dependencies.
- Empirical evaluation showing that our framework enables a more effective form of redundant load elimination.

II. Running Example

Figure 1 shows a snippet of a C function that will serve as a running example throughout the paper. Suppose we want to optimize the program by replacing the stores to the array `Y` with a single vector store. This is only possible if the two stores are independent; however, any sound static dependence analysis will conclude that the stores are dependent for the following reasons. First, there is a dependence chain from the last store (`Y[1] = 0`) to the load (`*X`) because the pointers `X` and `Y` may alias, and then to the first store (`Y[0] = 0`). Second, the function call also prevents vectorization because it may read and write arbitrary memory.

Figure 2 shows a program equivalent to the running example. We can verify that the programs are equivalent with a case analysis of the two `if` conditions and the resulting traces of operations. Although the two programs have the same semantics, there is one key difference, namely that in the rewritten program, the stores we are interested in are independent. Section III discusses how our framework automatically performs this transformation for general programs to ensure that the specified sets of instructions are independent.

$$
\begin{aligned}
\mathit{fn} &::= \mathit{item}_1 : p_1, \ldots, \mathit{item}_n : p_n \\
\mathit{item} &::= \textbf{instruction} \mid \mathit{loop} \\
\mathit{loop} &::= \textbf{with } v_1 = \mu_1, \ldots, v_m = \mu_m \textbf{ do} \\
&\qquad \mathit{item}_1 : p_1, \ldots, \mathit{item}_n : p_n \\
&\quad \textbf{while } p_{cont} \\
\mu &::= \textbf{mu}(v_{init}, v_{rec}) \\
\phi &::= \textbf{phi}(v_1 : p_1, \ldots, v_n : p_n) \\
p &::= \textbf{true} \mid v \mid \overline{v} \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \\
v &::= \mu \mid \textbf{instructions} \mid \textbf{constant} \mid \textbf{argument}
\end{aligned}
$$

Fig. 3: Definition of predicated SSA. Our framework uses predicated SSA to represent the input program to simplify control-flow code motion.

```
store [Y + 0], 0    ; true
x = load X          ; true
c = cmp ne x, 0     ; true
call cold_func      ; c
store [Y + 1],  1   ; true
```

Fig. 4: Running example from Figure 1 represented in predicated SSA.

III. Approach

We first present the general workflow of our versioning framework. First, during the planning stage, the client optimization provides us with a set of instructions (or loops) that are possibly dependent. In response, our system infers *a versioning plan* that ensures their independence. Each versioning plan outlines a set of instructions (and loops) that requires versioning and specifies the set of conditions under which versioning should take place. The client can make multiple requests to make different sets of instructions (or loops) independent, which can result in multiple versioning plans. Finally, the client invokes a separate routine to *materialize* the inferred versioning plans. Afterwards, the optimization can use the instruction dependence guaranteed by the plans to perform its transformations.

Control-flow Code Motion and Predicated SSA. Control-flow code motion (i.e., moving instructions across basic blocks and branches) is a basic operation on which our versioning framework relies. To simplify this operation, we first convert the input program to predicated SSA form [4]. Figure 3 shows the definition of predicated SSA. In predicated SSA, each instruction (or loop) is annotated with a predicate indicating whether the instruction should be executed. With these predicates, the IR no longer uses branches or basic blocks and instead models the program as a hierarchy of nested loops, each of which contains a flat list of member instructions and loops. This representation simplifies global code motion because individual instructions and loops are no longer tied to any specific basic blocks and can be moved without affecting correctness (provided no dependencies are violated). Figure 4 shows the program represented in predicated SSA form.

$$
\begin{aligned}
c &::= p \mid \mathit{intersection} \mid c_1 \vee c_2 \\
\mathit{intersection} &::= \textbf{intersects}([m_1, m_2], [m_3, m_4]) \\
p &::= \textbf{control predicate} \\
m &::= \textbf{memory address}
\end{aligned}
$$

Fig. 5: Definition of dependence condition. A dependence exists either due to some predicate p (e.g., an instruction i depends on another instruction j only if j is executed) or because of the intersection of two memory ranges (e.g., two memory writes are dependent if they alias).

```
def c(i, j):
  # return the condition for when i depends on j directly
  if is_phi(i):
    for op, op_pred in cast_to_phi(i).operands():
      if op == j:
        return op_pred
    return Predicate.false()
  if is_select(i):
    select = cast_to_select(i)
    if select.true_val == j:
      return pred(i).and_(select.cond)
    if select.false_val == j:
      return pred(i).and_(select.cond, negate=True)
    return Predicate.false()
  if i in j.users():
    return Predicate.true()
  if not i.may_write() and not j.may_write():
    return Predicate.false()
  if i.is_instruction() and j.is_instruction():
    # If j executes at a more specific predicate than i
    if pred(j).implies(pred(i)) and pred(j) != pred(i):
      return pred(j)
    return Intersects(mem_location(i), mem_location(j))
  # deal with cases when either i or j is a loop
  insts1 = [i.mem_instructions()] if i.is_loop() else [i]
  insts2 = [i.mem_instructions()] if j.is_loop() else [j]
  conds = []
  for i1 in insts1:
    for i2 in insts2:
      if i1.may_write() or i2.may_write():
        conds.append(c(i1, i2))
  return Or(conds)
```

Fig. 6: Algorithm for determining the dependence condition between two nodes in the dependence graph i and j.

A. Dependence Condition

The traditional notion of program dependence can be too restrictive and imprecise for complex, real-world programs. For example, if the alias analysis cannot soundly disambiguate the pointers used by a pair of memory accesses (assuming at least one is a write), traditional dependence analysis simply concludes that there is a dependence. However, this dependence only exists at run time if the accessed memory locations actually intersect. For this reason, we propose a more refined approach, in which program dependence is labeled with the necessary condition for the dependence to occur at run time. We refer to this condition as the *dependence condition*. Figure 5 shows the definition of dependence conditions.

Given two instructions i and j from the dependence graph, we use the notation $c(i, j)$ to represent the dependence condition for i to directly depend on j. Figure 6 shows the algorithm calculating $c(i, j)$. We assume the input program is represented

in predicated SSA and use $pred(i)$ to denote the predicate associated with the instruction i. Our framework identifies dependence conditions as follows:

- A ϕ node `t = phi (x1: p1, ... , xn : pn)` only depends on the operand x_i if p_i is true at run time; i.e., $c(t, x_i) = p_i$.
- A select instruction `t = select x, a, b` depends on a if x is true; i.e., $c(t, a) = pred(a) \wedge x$.
- In other cases where an instruction i uses the result of another instruction j as an operand, then i depends on j unconditionally.
- A necessary condition for an instruction i to depend on j is if j is ever executed. Therefore, when we detect that j executes at a more specific predicate than j, we return j's predicate as the dependence condition.
- Two memory accesses (assuming at least one is a write) are dependent if their memory locations intersect.
- Assuming there are no use-def dependencies, two loops l_1 and l_2 are dependent if any of their member instructions are dependent; i.e., $c(l_1, l_2) = \bigvee_{i \in insts(l_1), j \in insts(l_2)} c(i, j)$. We define the dependencies between a loop and an instruction similarly.

For any two instructions i and j, we refer to the dependence $i \rightarrow j$ as an *unconditional dependence* if $c(i, j) = true$; otherwise, we call it a *conditional dependence*.

For some of the dependence conditions, our system eventually needs to generate run-time checks to verify that those conditions are indeed true. In this sense, a dependence condition implies a computation that, in turn, has its own dependencies. For instance, if a dependence condition contains intersection checks, then the condition itself depends unconditionally on the memory addresses involved in the intersections.

Graph Cut and Versioning. Figure 7 shows the dependence graph of the running example, where the conditional dependencies are represented as dotted edges and unconditional dependencies are represented as solid edges. Note that we can make the two stores independent (i.e., unreachable from one another) by eliminating the following conditional edges: (1) the conditional dependence from the load to the first store and (2) the dependence from the last store to the function call. This is a core operation that we use extensively throughout our framework.

More generally, given two sets of dependence graph nodes S and T (*which may intersect*), we need to identify a set of *conditional dependencies* whose removal makes S independent of T. This is equivalent to finding a *cut* over the dependence graph so that nodes in T are not reachable from S,[1] with the *cut-set*—edges crossing the partitioned sets—exclusively containing edges for conditional dependencies. To optimize for performance, our algorithm also reduces the cut-set because each edge in the cut-set represents a dependence condition

[1]Because S and T may intersect, we ignore trivial reachabilities of the form $s \rightarrow s$, where $s \in S \cap T$.

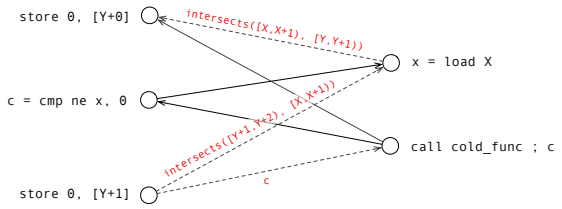


Fig. 7: Dependence graph of the running example. Conditional dependencies are represented as dotted edges and labeled with their dependence conditions. The function call depends unconditionally on the first store because the store has a stronger predicate (i.e., the execution of the call implies that of the store but not vice versa).

that must be eventually checked at run time. A smaller cut-set typically leads to better performance. If successful, the algorithm returns two sets: (1) the cut-set and (2) the source side of the cut that can reach T. Note that if the S and T are already independent, the algorithm returns two empty sets.

We find the cut with a reduction to min-cut, which works by constructing a flow graph such that there is a flow from the source to sink if and only if any of the nodes in S depends on T. A min-cut of the flow graph reveals the set of dependence edges whose removal would make the input instructions independent. Figure 8 shows the flow graph constructed for the running example. In general, we create the flow graph as follows:

1) We start with a depth-first search (DFS) to discover the dependencies of nodes in S, considering both conditional and unconditional dependencies.
2) We split each node discovered by the DFS into an in-node and an out-node. We then duplicate the dependence edges such that each dependence flows into an in-node and out of an out-node, with each pair of in- and out-nodes connected with a single auxiliary edge. Splitting the nodes is important because otherwise the sink will always be reachable from the source even when all the conditional edges are cut.
3) Next, we connect the source node to all of the out-nodes of S and all of the in-nodes of T to the sink node.
4) Finally, we set the capacity for each conditional edge to 1 and all the other edges to $n + 1$, where n is the total number of unconditional edges.

With the capacity thus set, if the min-cut is greater than n, we must disconnect at least one unconditional dependence edge to make the nodes independent, and versioning is infeasible. If the min-cut is less than n, then versioning is feasible, and we set the versioning conditions—which we speculate to all be false—as the union of the dependence conditions of the edges in the cut-set. With profile information, we can also set the capacity of each conditional edge as the likelihood of the corresponding dependence occurring at run time, thereby maximizing the pass rate of the run-time checks.

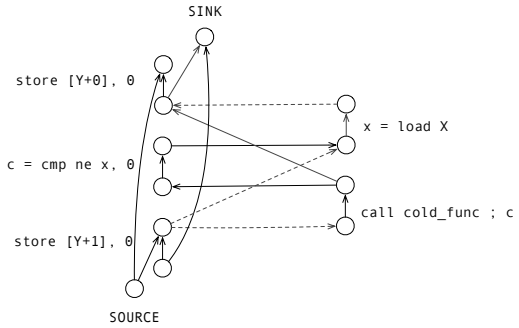


Fig. 8: Flow graph constructed for the running example to separate the two stores in the dependence graph. Dotted edges are conditional and have capacities of 1, while solid edges are unconditional and have capacities of 4.

B. *Versioning by Example*

We now discuss how our framework uses the aforementioned algorithm for finding dependence graph cuts as a subroutine to version the running example (Figure 4), using step-by-step examples. Section III-C discusses the general algorithm for versioning.

Versioning is more intricate than simply finding a cut in the dependence graph. It might be tempting to version the program directly given a cut by duplicating nodes on one side of the cut and using the dependence conditions from the cut-set as the versioning conditions. Nevertheless, this approach is generally unsafe because materializing a versioning plan involves generating code to check that none of the dependence conditions are true. These checks themselves are computations that can depend on the nodes being versioned, and when they cannot be computed because they depend on the code under versioning, we need to identify a *secondary* versioning plan to ensure that the initial set of versioning conditions is independent of the code that was intended to be versioned in the first place. We refer to the initial versioning plan as the *primary* versioning plan, and such a collection of plans—primary, secondary, etc—constitutes a single *nested* versioning plan.

The running example (Figure 4) is one such example that requires a nested versioning plan. To start, we aim to make the two stores independent, which entails finding a cut that separates the two stores. Figure 9 shows the flow graph and the min-cut. The cut has the following dependence edges in its cut-set (marked with ❌):

- The dependence from the second store (`store [Y+1], 0`) to the function call, which has the dependence condition c; i.e., the dependence only exists if c is true.
- The dependence from the load (`x = load X`) to the first store (`store [Y+0], 0`), which has the dependence condition $intersects([X, X+1), [Y+0, Y+1))$; i.e., the dependence only exists if the addresses `X` and `Y+1` alias.

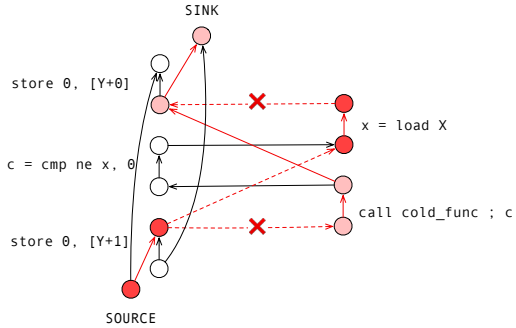


Fig. 9: Primary cut to separate the two stores in the dependence graph. Nodes and edges with positive flow are highlighted in (different shades of) red. ● represents a node in the source-side cut. ● represents a node in the sink-side cut. Edges in the cut-set are marked with ❌.

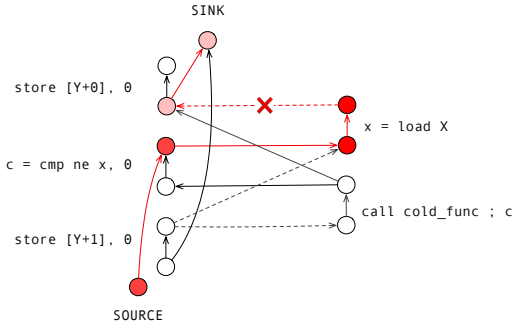


Fig. 10: Secondary cut to make the comparison (`c`) independent of the stores.

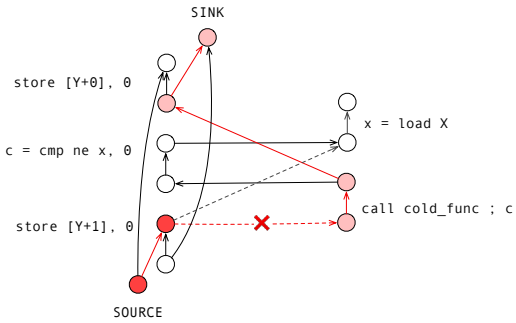


Fig. 11: Primary cut from Figure 9 updated to reflect that the dependence from the load to the first store is removed by the secondary cut (Figure 10).

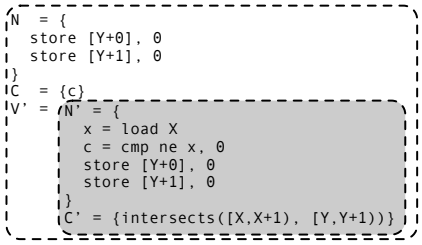


Fig. 12: Final nested versioning plan derived from the cuts shown in Figures 10 and 11. N (and N') are the nodes that will be versioned and C (and C') are the conditions under which they will be versioned.

We focus on the conditional dependence that involves the conditional function call. It has a dependence condition of c, which itself depends (transitively) on the first store (because `X` may alias with `Y`). Because we must be able to evaluate c before both of the stores, the plan is not materializable. More generally, a plan is materializable if its versioning conditions are independent of its versioned nodes. Therefore, we proceed to infer a secondary versioning plan that makes c independent of the stores.

Figure 10 shows the secondary cut. The cut has exactly one edge in the cut-set, the dependence edge from the load to the first store (`x = load X` $\rightarrow$ `store [Y+0], 0`), which has the dependence condition $intersects([X, X+1), [Y+0, Y+1))$. We do not need a tertiary plan here because this dependence condition only depends on the pointers X and Y and no other instructions.

Given the secondary plan, we finish by updating the initial flow graph constructed for the primary plan. The updated flow graph and (primary) cut are shown in Figure 11. Note that we delete the dependence edge from the load to the first store (`store [Y+0], 0`) because this dependence has been eliminated by the secondary plan. Having considered the removal of this dependence, we update the source-side cut to include only the second store (`store [Y+1], 0`). Figure 12 shows the final versioning plan—constructed based on the cuts we just computed—making the two stores independent. Essentially, for a given cut, we perform versioning on stores as well as any nodes on the source side that can reach the stores.

C. Inferring Versioning Plans

We are now ready to formalize the example procedure for inferring a versioning plan given a set of instructions that we wish to make independent. A *versioning plan* is an abstraction that allows us to represent a subprogram that we want to version and the condition under which it should be executed without explicitly transforming the program. More specifically, a versioning plan is a three-tuple $V = (N, C, V')$, where:

- N is the set of dependence graph nodes that we want to version, with each node representing either an instruction or a loop.
- C is the set of versioning conditions that we will assert to be false at run time, and if any of them is true, execution will proceed to the fallback path.
- V' is an optional *secondary versioning plan*, which becomes necessary if some of the dependence conditions (C) transitively depend on a subset of the versioned nodes (N), introducing circular dependencies that make code generation impossible.

Figure 13 shows the algorithm for inferring a nested list of versioning plans so that a given list of instructions is independent, with the initial primary plan ensuring that the input instructions are independent and each subsequent secondary plan ensuring that a previous primary plan is materializable (i.e., ensuring that the dependence conditions of the primary version do not depend on the versioned nodes).

The algorithm for inferring versioning plans is recursive and takes two sets of nodes as input: `nodes` and `input_nodes`. Each recursive call returns a versioning plan that—once materialized—ensures that no node in `nodes` depends on `input_nodes`. The algorithm starts by finding a cut of the dependence graph that separates `nodes` from `input_nodes`. The versioning conditions are set as the union of the conditions of the dependencies in the cut-set. If the versioning conditions are all false, then it implies that `nodes` are independent of `input_nodes`.

At run time, the transformed program must verify that all of these versioning conditions are false before executing the input instructions (`input_nodes`) as vector instructions. Based on this execution strategy, the versioning conditions must not depend on the `input_nodes`; otherwise, those conditions cannot be evaluated before `input_nodes`. Consequently, instead of directly constructing a primary versioning plan according to the initial cut, we need to recursively run the algorithm to ensure that the versioning conditions are independent of `input_nodes` (lines 11-21). Before we attempt to find a secondary versioning plan recursively, it is crucial to check that none of the versioning conditions *directly* uses any of the `input_nodes` (at line 16) because otherwise the versioning conditions would always depend on the `input_nodes` *unconditionally*.

With the secondary plan in hand, we then update the cut to take into account the dependence edges that no longer exist due to secondary versioning. Alternatively, we could compute this cut from scratch to account for the dependence edges removed by secondary versioning. Finally, the algorithm constructs the primary versioning plan to version only the source side of the cut that can reach `input_nodes` and the `input_nodes` themselves.

Although the recursive call only constructs a secondary plan guaranteeing that the versioning conditions (of a primary plan) are independent of the `input_nodes`, the secondary plan also implicitly guarantees that the versioning conditions are independent of *all versioned nodes*, which is necessary for a plan to be materializable. In other words, if a versioning condition c does not depend on any of the input instructions, then c must also be independent of any node n versioned by

the primary plan. We can prove this by contradiction. Suppose, for contradiction, that a versioning condition c depends on such a node n. Recall that n, by definition, is either (1) a node on the source side of the cut that can reach input instructions (`input_nodes`) or (2) one of the input instructions. Either case would imply that there is a path from n to some of the input instructions, leading to the contradiction that c depends on n and thus an input instruction.

The versioning plan inference algorithm always terminates, which we show by fixing a topological ordering for all the nodes in the dependence graph. The original program order is one such ordering, which assigns a unique number, $order(.)$, to each dependence graph node so that if a node s depends on another node t, then $order(s) > order(t)$. Observe that if the following condition is true, then the recursion terminates (because if $order(s) \leq order(t)$, then s cannot depend on t):

$$\max_{s \in \texttt{nodes}} order(s) \leq \min_{t \in \texttt{input\_nodes}} order(t)$$

It suffices to show that the ordering of the `nodes` is always decreasing, which is the case because the *operands* of the versioning conditions—i.e., the nodes that we pass to the recursive call (line 19)—are always ordered before at least one of the `nodes`. To see why this is true, recall that each versioning condition is the dependence condition of one of the edges in the cut-set. Let $s \rightarrow t$ be an edge in the cut-set and x be an operand of the dependence condition $c(s,t)$. Following the definition of dependence conditions, there are two cases here: (1) $c(s,t)$ involves an intersection check, and x is some memory address used by s or t; and (2) $c(s,t)$ involves a predicate and $x = pred(t)$. Either case implies that s depends on x, and consequently x is ordered before s and at least one of the `nodes`.

D. Materializing Versioning Plans

Figure 14 shows the algorithm for materializing versioning plans. The algorithm recursively materializes any secondary plans before materializing the primary plans. It uses a table that maps each instruction (or loop) to the set of dependence conditions under which the instruction (or loop) is versioned. Given that an instruction (or loop) may be involved in more than one versioning plan, its entry in this table is the union of all the conditions in the involved versioning plans.

Figure 15 shows how the versioning plan of the running example (Figure 12) is materialized in steps. First, the secondary plan is lowered (Figure 15a) by duplicating all of the versioned nodes and guarding them with checks computing the versioning conditions. After the secondary plan is lowered, the primary plan is lowered similarly (Figure 15b). It is important to note that the materialization algorithm hoists the comparison as well as the load (of `X`) before stores (versioned by the primary plan) to compute the primary versioning condition, whereby this hoisting is enabled by the secondary versioning. Figure 15c shows the final versioned program further lowered to SSA (with control flow).

```
1  def infer_versioning_plan(nodes, input_nodes):
2    # Find a cut in the dependence graph that makes
3    # `nodes` independent from `input_nodes`
4    cut = find_cut(nodes, input_nodes)
5    # Abort if such a cut does not exist
6    if cut is None:
7      return None
8
9    dep_conds = set_union(c(e.src, e.dst)
10                         for e in cut.cross_edges)
11   # Find a secondary versioning to make sure that
12   # we can evaluate the dependence conditions
13   # before the versioned nodes.
14   cond_operands = set_union(operands(dep_cond)
15                             for dep_cond in dep_conds)
16   if set_intersect(cond_operands, input_nodes):
17     return None
18   secondary_plan =\
19     infer_versioning_plan(cond_operands, input_nodes)
20   if secondary_plan is None:
21     return None
22
23   # Update the cut according to the secondary plan.
24   # This is equivalent to:
25   #   1) updating the dependence graph to remove
26   #      edges that no longer exists because
27   #      of the secondary plan and
28   #   2) rerunning `find_cut` on the updated graph.
29   cut = update_cut(cut, secondary_plan)
30   return VersioningPlan(
31              nodes=cut.source_nodes + input_nodes,
32              versioning_conds=dep_conds,
33              secondary=secondary_plan)
34
35 def infer_version_plans_for_insts(insts):
36   return infer_versioning_plan(
37              nodes=insts, input_nodes=insts)
```

Fig. 13: Algorithm for inferring a versioning plan that makes a list of instructions independent. Lines 11-21 deal with secondary versioning.

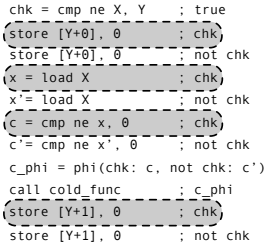
More generally, the algorithm for materializing the versioning plans works as follows. First, we emit a check for each unique set of dependence conditions to ensure they are all false, after having also hoisted the dependencies of the check before the versioned nodes. This is feasible because in the planning phase, we have ensured that the conditions do not depend on the versioned nodes. After emitting the checks, we clone the nodes and strengthen their control predicates so that the original copy executes if its check passes and the clone executes if the check fails.

Versioning may break the invariant that all definitions should dominate their uses. Therefore, we must update the instructions that use the versioned nodes as operands. More specifically, let i_{def} be an instruction that we want to version, i_{user} one of its users, and i'_{def} the cloned version of i_{def}. After versioning, i_{def} might no longer dominate i_{user}. This occurs when i_{user} is either not versioned or versioned under a set of conditions not implied by the versioning conditions of i_{def}. In this case, we need to replace all uses of i_{def} with an auxiliary phi node that takes the value of i_{def} if the check passes and i'_{def} if the check fails. Lines 49-57 show the algorithm for restoring the def-use dominance for a versioned instruction. Similarly, we also restore def-use dominance for the live-out instructions of versioned loops.

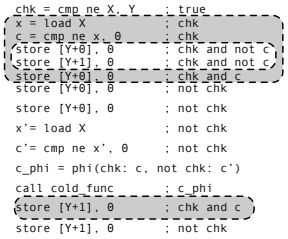
```
def materialize_plans(versions):
  # lower the secondary versions
  secondaries = [version.secondary
    for version in versions
    if version.has_secondary()]
  if len(secondaries) > 0: materialize_plans(secondaries)
  # Build a table that maps each instruction/loop
  # to its versioning conditions
  versioning_table = build_versioning_table(versions)
  # Mapping each cloned instruction/loop to its clone
  clones = {}
  # Mapping each cloned instruction to a phi that
  # joins the original and clone
  versioning_phis = {}
  for inst_or_loop, conditions in versioning_table:
    # Emit a check to return true iff *none* of
    # the conditions are true.
    # Also take care to hoist any dependencies that
    # exists *after* this inst/loop.
    check = emit_check(conditions,
                       insert_before=inst_or_loop)
    clone = inst_or_loop.clone(
                       insert_after=inst_or_loop)
    clones[inst_or_loop] = clone
    p = pred(inst_or_loop)
    # Predicates indicating if the check passes/fails
    success_p = PredicateAnd(p, check)
    failure_p = PredicateAnd(p, check, negate=True)
    # The original computation runs if the check passes
    inst_or_loop.update_predicate(success_p)
    # The cloned computation runs if the check fails
    clone.update_predicate(failure_p)
    if inst_or_loop.is_instruction():
      inst = inst_or_loop.cast_to_inst()
      phi = Phi.make(
              {success_p: inst, failure_p: clone},
              insert_after=clone)
      versioning_phis[inst] = phi
    else:
      # Process the loop instructions similarly
      ...

  # Change the cloned instructions to
  # use the correct operands
  for inst_or_loop, conditions in versioning_table:
    if inst_or_loop.is_instruction():
      inst = inst_or_loop.cast_to_inst()
      cloned_inst = clones[inst]
      for user in inst.uses():
        if conditions.subset_of(versioning_table[user]):
          cloned_user.replace_uses_of(inst,
                                      with=cloned_inst)
        else:
          phi = versioning_phis[inst]
          user.replace_uses_of(inst, with=phi)
          if user in clones:
            clones[user].replace_uses_of(inst, with=phi)
    else:
      # Process the uses of the loop live outs similarly
      ...
  # DCE on the versioning phis
  for phi in versioning_phis:
    if phi.has_no_uses():
      phi.erase()
  # Some phi's incoming predicates become dead
  # (i.e., always false).
  # Replace those operands with a placeholder value.
  for inst_or_loop, conditions in versioning_table:
    if inst_or_loop.is_phi():
      phi = inst_or_loop.as_phi()
      for op, op_pred in phi.operands():
        if is_implied(op_pred, conditions):
          phi.set_operand(UNDEFINED)
```

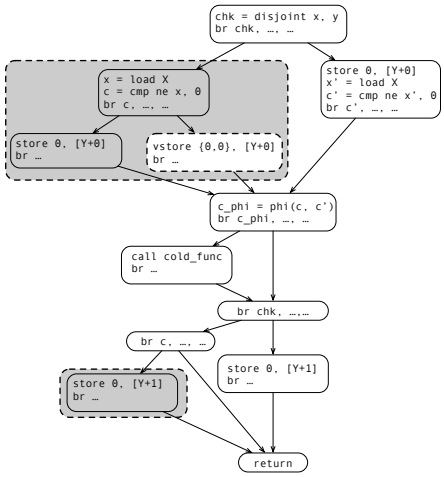
Fig. 14: Algorithm for versioning a program according to a set of inferred versioning plans. For simplicity, the pseudocode emits the versioning checks for each instruction (or loop) from scratch without regard to redundancy; our implementation only emits one check for each unique set of conditions.



(a) Running example with the secondary versioning plan materialized. Because the comparison (c) is versioned, we insert a ϕ node to join the two versions of the comparison before it is used (as a predicate) by the function call.



(b) Running example with the secondary and primary versioning plans materialized.



(c) Running example after versioning and conversion back to SSA with control flow.

Fig. 15: Running example with the versioning plan (from Figure 12) materialized. Instructions versioned by the primary plan are surrounded with , and those instructions versioned by the secondary plan are surrounded with . We materialize the secondary plan before the primary plan.

IV. Implementation

We implemented the proposed versioning framework as an LLVM [8] library with $4,982$ lines of C++ code. The interface of the library has two functions: the first performs versioning plan inference by taking a list of instructions (or loops) and returning a versioning plan that makes those instructions independent; this function may report that it is infeasible to infer such a plan; and the second function materializes a given list of versioning plans into actual IR instructions and control flow. In this section, we discuss some important implementation choices for the framework to work in practice, for correctness and performance.

A. Optimizing Dependence Conditions

Before materializing the versioning plans, we additionally allow the user to run some optimizations on their versioning plans to reduce the cost of run-time checks. More specifically, each memory intersection requires us to emit a sequence of pointer arithmetic and comparisons to perform the check. We have three optimizations to reduce the cost of performing the intersection checks: *redundant condition elimination*, *condition coalescing*, and *condition promotion*.

Redundant Condition Elimination. Some intersection checks are logically equivalent (but syntactically different). For example, we can see with basic algebraic manipulation that if $[a, a + 10]$ intersects with $[b, b + 2]$, then $[a + 100, a + 110]$ also intersects with $[b+100, b+102]$. In general, two intersection checks $intersects(r_a, r_b)$ and $intersects(r_x, r_y)$ (where $r_.$ are memory ranges) are equivalent if $offset(r_x, r_a) = offset(r_y, r_b)$ or if $offset(r_x, r_b) = offset(r_y, r_a)$, where $offset)(.,.)$ provides the offset between two memory ranges. The offset between two ranges is undefined when their lower bounds and upper bounds differ by two different numbers; when this occurs we simply conclude that the two intersection checks are not equivalent. To perform redundant condition elimination, we first partition the dependence conditions into equivalence classes and then replace each dependence condition with an arbitrarily chosen representative from its equivalence class.

Condition Coalescing. Condition coalescing replaces two different range checks with a single check that overapproximates the original checks. The final coalesced check might be less precise yet cheaper to compute. For example, we can coalesce $intersects([a, a + 10), [b, b + 10))$ and $intersects([a + 20, a + 30), [b + 40, b + 50))$ into a single check $intersects([a, a + 30), [b, b + 50))$. Because condition coalescing results in less precise checks (that fail more often), we perform condition coalescing after redundant condition elimination.

Condition Promotion. Condition promotion generalizes a range check to be loop-invariant so that the promoted check can be hoisted outside a loop. Consider the intersection check $intersects([a+i, a+i+2), [b+i, b+i+4))$, where i is a loop induction variable. In this case, we can promote the check to the equivalent check $intersects([a, a+2), [b, b+4))$, which no longer involves the induction variable and is therefore loop-invariant. This type of promotion is precise; i.e., a promoted check passes if and only if the original check passes. We also perform a second type of promotion that is imprecise. For example, we can promote the check $intersects([a+i, a+i+1), b[b + 4 * i, b + 4 * i + 4))$ to the check $intersects([a, a + N + 2), [b, b + 4 * N + 4))$, where N is the trip count of the loop. We only perform this type of promotion when the trip count of the loop is known before the loop is executed. Because this kind of promotion is imprecise, it's only applied to intersection checks involving different memory objects.

B. Aliasing Annotation

Although our versioning plan materialization algorithm generates the requisite checks and control flow to guarantee that the user-supplied instructions (or loops) are independent, the independence of these instructions is implicit and unlikely to be detected by LLVM's built-in alias and dependence analysis. Thus, our implementation additionally annotates the versioned instructions with LLVM's scoped noalias metadata [1].

Using LLVM's scoped noalias system provides two benefits. First, it allows us to reuse LLVM's existing alias analyses and be compatible with any client optimization that uses those analyses. Second, LLVM's instruction cloning utility preserves these noalias annotations, which is useful when an instruction is cloned multiple times due to nested versioning.

V. Evaluation

We evaluated the proposed versioning framework by using it for two optimizations: redundant load elimination and SLP vectorization. We show that our framework makes both optimizations more effective with relatively minimal changes to their implementations.

Experimental Setup. We ran all experiments on a machine with an Intel® Xeon® Platinum 8124M CPU running at 3.00GHz. All benchmarks were single-threaded, run sequentially, and pinned to a specific core with `taskset`. All benchmarks were run 30 times and the median run times were used for reporting. Unless otherwise specified, we used LLVM's -O3 pipeline as the baseline, which used LLVM's loop and SLP vectorizers with the former using LLVM's loop versioning implementation.

A. Versioning for SLP Vectorization

To demonstrate the effectiveness of our versioning framework, we modified an existing SLP vectorizer to support versioning. Before discussing how we integrated versioning with SLP vectorization, we first provide some background.

Background. Larsen and Amarasinghe [7] proposed SLP vectorization as an alternative to loop vectorization. SLP vectorization works in two stages, whereby the vectorizer first uses a heuristic to identify *packs* of independent instructions that are profitable candidates for vectorization, and then it finishes by replacing these packs of instructions with equivalent vector instructions. The vectorizer might also need to emit extra data-movement instructions to gather the operands of the vector instructions; for instance, if a vector instruction uses some array elements out of order, the vectorizer must emit a shuffle instruction to permute those elements.

While loop versioning is natural to apply for loop vectorization, there is no equivalent versioning scheme for SLP vectorization. With the unit of vectorization being individual vector packs and considering the diversity of the different SLP packing heuristics [11, 12, 13, 14, 15], a versioning scheme for SLP vectorization must be more fine-grained to fully take advantage of its flexibility.

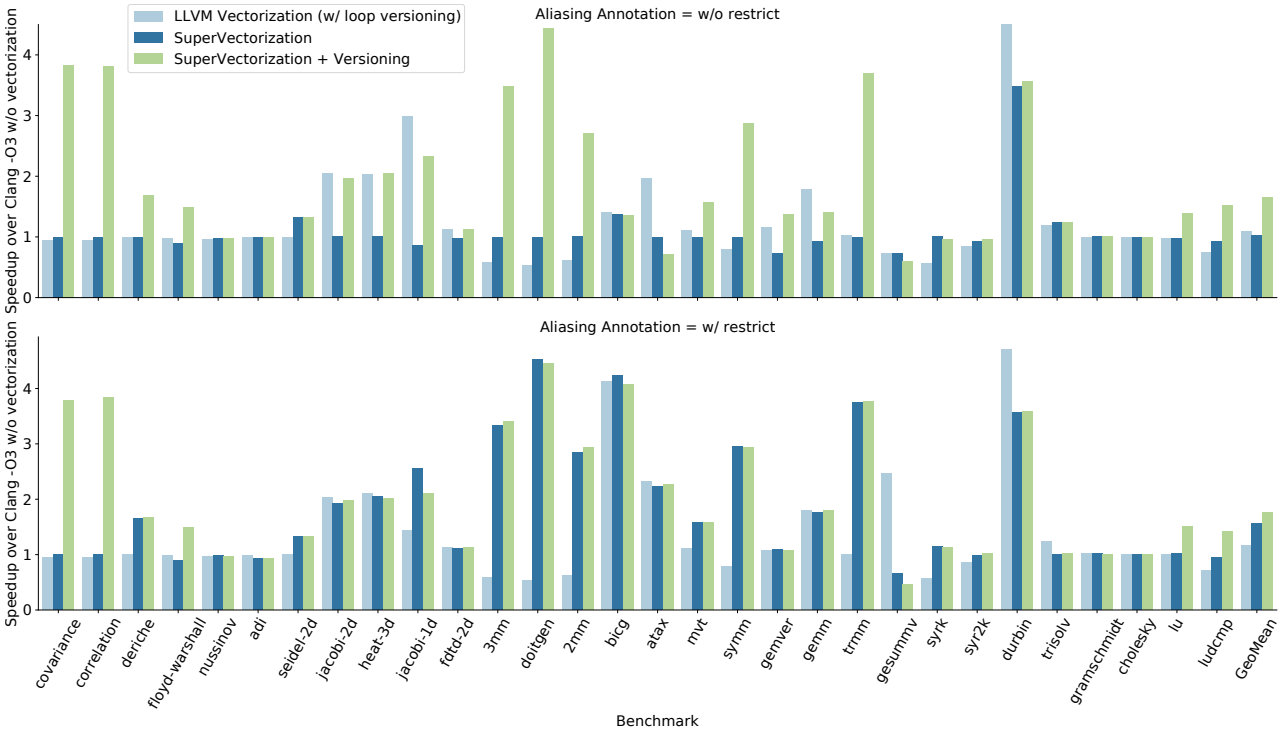


Fig. 16: Speedup on PolyBench over LLVM -O3 without vectorization (higher is better).

1) Versioning for SLP Vectorization: We added support for versioning to SuperVectorization [4], an extension of SLP vectorization to handle complex data-dependent control flow. This vectorizer can pack instructions across different basic blocks or even loops that might have divergent runtime trip counts. Due to these complex control-flow patterns, their code-generation algorithm is more involved than a more traditional SLP vectorizer. Nonetheless, we added versioning to their vectorizer without having to modify any of their code-generation machinery.

We made only two changes to SuperVectorization to enable versioning. First, we modified its implementation to allow packing instructions that are conditionally independent. Its system originally had a filter pass to reject any packs involving dependent instructions for vectorization. We modified this pass to forward such packs to our versioning plan inference routine and accept a pack for vectorization if we can infer a versioning plan. At the end of the packing stage (but before vector code generation), our system ends up with a list of versioning plans. The second change was to materialize the list of versioning plans before performing vector code generation. No other changes were necessary, and their original vector code generator works out of the box.

2) Results: We evaluated the modified SuperVectorization on two benchmark suites: PolyBench [16] and TSVC [3]. We used the same optimization pipeline as the LLVM `-O3` optimizations, aside from the fact that we ran Chen et al.'s [4] vectorizer instead of LLVM's loop and SLP vectorizers (which are disabled in our pipeline). Note that LLVM's loop vectorizer uses loop versioning, which is the same setting that Chen et al. [4] used in their original evaluation.

PolyBench. Figure 16 shows the results on PolyBench. We ran the experiments in two settings, with the `restrict` keyword enabled and disabled. Because LLVM does not perform interprocedural alias analyses, without the `restrict` keyword, LLVM's dependence analysis conservatively assumes that all input arrays in the PolyBench kernels may alias.

With the `restrict` annotation turned off, SuperVectorization with versioning obtains a geomean $1.65\times$ over LLVM's scalar (`-O3`) pipeline and $1.50\times$ faster than LLVM's full `-O3` pipeline (with vectorization enabled). With `restrict` turned on, SuperVectorization together with versioning obtains a $1.76\times$ speedup over LLVM's scalar pipeline and $1.51\times$ faster than LLVM with vectorization; whereas SuperVectorization without versioning is $1.57\times$ faster than LLVM's scalar optimization and $1.35\times$ faster than LLVM with vectorization.

In sum, even with full aliasing information, versioning enables SuperVectorization to vectorize five more benchmarks (correlation, covariance, floyd-warshall, lu, and ludcmp), boosting SuperVectorization's performance by $1.12\times$ on average. All but one of the five benchmarks have loop iterations that are independent but involve triangular iteration space, which is too complex for LLVM's dependence analysis to reason. With versioning, SuperVectorization effectively performs dependence analysis dynamically.

```
1  for (k = 0; k < _PB_N; k++)
2    for (i = 0; i < _PB_N; i++)
3      for (j = 0; j < _PB_N; j++)
4        path[i][j] = path[i][j] < path[i][k] + path[k][j]
5                   ? path[i][j] : path[i][k] + path[k][j];
```

Fig. 17: Snippet of the floyd-warshall benchmark from PolyBench. The read-write conflicts arising from the in-place updates on the `path` array prevents auto-vectorization. Our versioning scheme enables vectorization by checking for this conflict dynamically at run time.

Figure 17 shows a code snippet of the floyd-warshall benchmark from PolyBench. Neither SuperVectorization (without versioning) nor LLVM can vectorize the floyd-warshall benchmark because the benchmark performs in-place updates on the output buffer, leading to a dependence from the read of `path[i][k]` to the write of `path[i][j]`. However, this read-write conflict does not occur on every loop iteration. Our versioning algorithm allows SuperVectorization to safely vectorize by checking for this dependence dynamically and executing the vectorized code in the absence of this dependence.

Figure 18 shows floyd-warshall optimized by our versioning algorithm. Notice that the code does not check the dependencies between the write `path[i][j]` and the read `path[k][j]`. Although this dependence blocks traditional auto-parallelization, it does not prevent SLP vectorization by itself and is left out of run-time checking by our versioning algorithm. We illustrate how our algorithm comes to this decision by unrolling the code by a factor of 2 (some other unroll factor also works) as follows:

```
... = path[k][j]
... = path[i][k]
... = path[i][j]
path[i][j] = ...
... = path[k][j+1]
... = path[i][k]
... = path[i][j+1]
path[i][j+1] = ...
```

We can see that the write to `path[i][j+1]` may depend on the reads of `path[i][j+1]` and `path[k][j+1]` (we ignore the dependence from `path[i][j+1]` to `path[i][k]` because it is in the cut), neither of which depend on `path[i][j]`. We can therefore conclude that the writes to `path[i][j]` and `path[i][j+1]` are independent and can be packed into a single vector store. One can follow this process and conclude that the other reads are also independent and can be packed, provided the accesses to `path[i][j:...]` and `path[k][j...]` are disjoint (which we check at run time).

TSVC. Figure 19 shows the results of TSVC, a comprehensive suite of 151 loops for evaluating vectorization algorithms. Without versioning, SuperVectorization is $1.09\times$ (geomean) faster than LLVM, and with versioning, SuperVectorization is $1.17\times$ faster. Versioning enables SuperVectorization to vectorize thirteen more benchmarks, improving the overall

```
1  for (k = 0; k < _PB_N; k++) {
2    for (i = 0; i < _PB_N; i++)
3      for (j = 0; j < _PB_N; j += VL) {
4        if (disjoint(&path[i][j:j+VL], &path[i][k])) {
5          // vector code
6        } else {
7          // scalar code
8        }
9    }
```

Fig. 18: Sketch of the code generated by our versioning algorithm for floyd-warshall. `VL` is the vector length chosen by the vectorizer.

performance of SuperVectorization by $1.08\times$ on average. Figure 20 shows a code snippet from s281. Neither original SuperVectorization nor Clang's vectorizers can vectorize s281 because the code has a read-write conflict between some but not all iterations. The authors of TSVC originally envisioned a transformation that partitions the loop iteration space and vectorizes the subspace where vectorization is safe. In contrast, our versioning framework enables vectorization by precisely identifying (at run time) those iterations where such conflicts are absent and executing those iterations in parallel.

Figure 21 shows a snippet of the s258 benchmark from TSVC. Our versioning framework enables vectorization by speculating that successive iterations of `a[i] > 0` are true at run time. We did not observe speedup (or slowdown) in TSVC's original setup because with versioning, the run time is data-dependent. Consequently, we ran another set of experiments where we initialized the array `a` randomly so that more than 99% of the entries are positive. In this setup, the vectorized code is $2.0\times$ faster than the scalar version. In a separate experiment, we further modified the benchmark so that all of the arrays are declared as function parameters (instead of global variables), forcing the compiler to assume the arrays may alias. In this setup, one level of versioning is no longer sufficient; to generate code to speculate that `a[i] > 0` for successive iterations, we must hoist the comparisons and the load of `a[i]` past the stores to the arrays `b` and `e`, which is only safe if a does not alias either `b` or `e`. We observe similar speedups even with two levels of versioning for this example because our implementation is able to hoist the alias checks outside the loop and amortize the overhead of the checks.

B. Versioning for Redundant Load Elimination

To further demonstrate the effectiveness of our versioning framework, we used it to perform redundant load elimination, a basic optimization implemented by modern compilers. A load is redundant if (1) it is dominated by a prior load on the same address and (2) there are no intervening writes between them. In practice, the latter condition often prevents compilers from eliminating many redundant loads because between two redundant loads, there are often some spurious writes—stores on plausibly aliased addresses or even function calls—that force the compiler to assume, conservatively, the loads are not redundant.

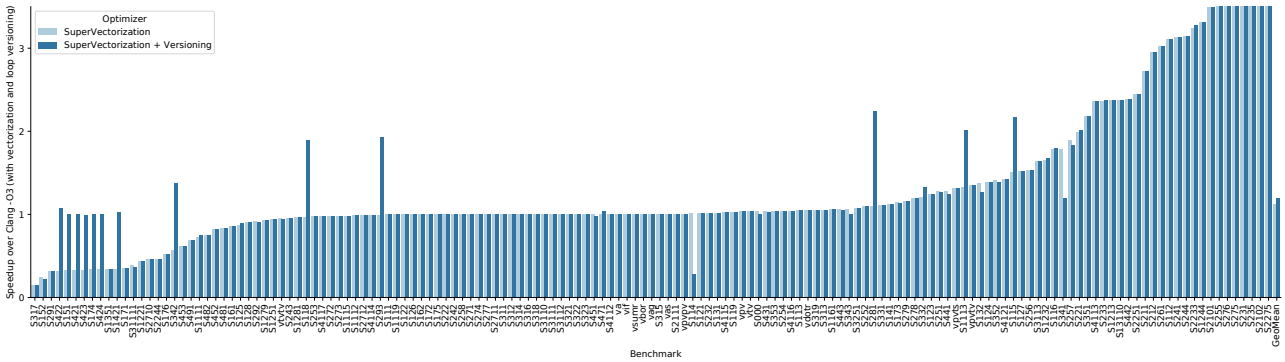


Fig. 19: Speedup on TSVC over LLVM -O3 (higher is better). Note that LLVM's loop vectorizer uses loop versioning.

```
1    for (int i = 0; i < LEN; i++) {
2      x = a[LEN-i-1] + b[i] * c[i];
3      a[i] = x-(float)1.0;
4      b[i] = x;
5    }
```

Fig. 20: s281 from TSVC. There is a read-write conflict on the array a, but only in some iterations (around the middle of the array). Our versioning framework generates code that identifies precisely the iterations that are safe to vectorize at run time.

```
1    float s = 0.;
2    for (int i = 0; i < N; i++) {
3      if (a[i] > 0.) {
4        s = d[i] * d[i];
5      }
6      b[i] = s * c[i] + d[i];
7      e[i] = (s + (float)1.) * aa[i];
8    }
```

Fig. 21: s258 from TSVC. Without versioning, the loop-carried dependence around `s` prevents vectorization.

We can use our versioning framework to eliminate more redundant loads by viewing redundant loads from the perspective of independence: A set of loads (assuming one load dominates all the others) is redundant if they are all independent. This leads to a simple implementation of redundant load elimination that is also more aggressive than those implemented by existing compilers:

1) Collect groups of loads on the same addresses (and of the same data types). Each group should have at least one load whose execution is implied by the other loads, and we call such a load the *leader* of the group.
2) If the loads in a group are not already independent, attempt to infer a versioning plan for the group so that they become independent, and remove the group for optimization if versioning is infeasible.
3) Materialize the inferred versioning plans.
4) Hoist the *leader* of each group before all the other loads—such hoisting is safe because we have established that the loads are independent—and replace the uses of the other loads with the leader.

We implemented this optimization as an LLVM pass with 153 lines of C++.

Figure 22 shows the result of running the proposed redundant load elimination pass on the SPEC 2017 Floating Point benchmark suite. Our optimization leads to major speedups for `lbm_r` (6.4%) and `blender_r` (4.7%), with the geomean speedup being 1.2%.

We also collected performance counters for the benchmarks. With versioning, redundant load elimination eliminated 4.8% (geomean) of the loads executed dynamically, at the expense of executing 5.5% (geomean) more branches at run time. Our optimization also enabled more optimizations downstream in the pipeline: loop-invariant code motion (LICM) hoisted 6.4% more instructions and global value numbering (GVN) deleted 8.5% more instructions.

VI. Related Work

Loop Versioning. Existing loop versioning implementations, such as the ones used by GCC and Clang, focus on pointer disambiguation and require predictable memory access patterns to overapproximate the accessed ranges of memory.

To address this limitation, researchers have proposed using specialized memory allocators that can efficiently check whether two pointers can point to the same allocated object [2, 5, 6]. With such allocators, loop versioning can be applied to loops—regardless of their memory access patterns—by querying these allocators whether the base pointers of different accesses point to the same objects.

This allocator-based loop versioning scheme is more efficient than our approach because it can disambiguate pointers *outside* loops with irregular memory accesses. In contrast, to check the dependencies for such loops, our approach must perform the checks *within* those loops.

	namd_r	parest_r	povray_r	lbm_r	blender_r	imagick_r	nab_r	**GeoMean**
Speedup	0.5%	−0.5%	−1.7%	6.4%	4.7%	0.0%	0.0%	1.2%
Loads eliminated	1.2%	−0.43%	−0.21%	26%	0.69%	−0.22%	2.7%	4.8%
Branches increase	3.4%	1.1%	1.3%	3.5%	−0.73%	0.0%	2.2%	5.5%
Extra instructions hoisted by LICM	50%	0.15%	0.1%	0.0%	1.2%	0.9%	0.0%	6.4%
Extra instructions deleted by GVN	0.59%	9.6%	1.3%	3.3%	19%	7.1%	7.7%	8.5%
Code size increase	1.5%	0.73%	1.2%	12%	0.88%	0.51%	0.00%	2.3%

Fig. 22: Result of running redundant load elimination (with versioning) on the SPEC 2017 Floating Point benchmark suite.

On the other hand, our approach can rule out intra-object dependencies—e.g, those arise from in-place updates—whereas theirs cannot. Our approach is also more fine-grained and decoupled from specific optimizations, whereas their approach uses hand-designed and optimization-specific rules to decide which dependencies should be checked. Furthermore, our approach works in the absence of loops and applies to non-loop optimizations.

Speculative Optimization. Our framework is also related to Lin et al.'s [9] work on performing speculative partial redundancy elimination (PRE) in the presence of rare dependencies. Their work relies on the architectural speculation support provided by IA-64. Our approach does not perform speculation and targets more conventional architectures without speculation capabilities.

Finally, our work is related to Sujon et al.'s [17] work on speculative loop vectorization, which was proposed to deal with loops with biased branches that incur rare loop-carried dependencies that prevent vectorization. Their approach works by identifying sets of paths where vectorization is safe, vectorizing the loop, and then generating run-time checks to guard vectorized code in case any other paths are taken. If the checks fail, the loop restarts execution in scalar mode for the next several iterations. Their approach relies on explicitly enumerating all possible CFG paths as candidates for speculation. Because there can be an exponential number of paths within a given CFG, their approach must consider exponentially many paths or up to a predetermined abort threshold. Our approach does not suffer from exponential blow-ups because it reduces the problem of finding the set of speculation (versioning) conditions to min-cut, which has an efficient solution.

VII. Conclusion

We have introduced a general framework for fine-grained versioning and demonstrated its effectiveness with case studies on applying our framework to redundant load elimination and SLP vectorization. Our redundant load elimination is simple and effective in removing many redundant loads that static analyses fail to detect. With our versioning framework, an existing implementation of SLP vectorization has been extended with minimal changes to perform versioning, achieving speedups that often outperform LLVM's vectorizers that use loop versioning.

Acknowledgments

We thank Willow Ahrens, Manya Bansal, Ajay Brahmakshatriya, Teodoro Fields Collin, Daniel Donenfeld, Katherine Mohr, Changwan Hong, Mit Kotak, Katsumi Okuda, Francesco Peverelli, Jaeyeon Won, and anonymous reviewers for their valuable comments and suggestions. Our work is supported by Intel Corporation and the National Science Foundation under Grant CCF-2217064; the Defense Advanced Research Projects Agency Awards HR0011-23-C-0101 and HR0011-23-0139. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] "LLVM language reference manual," https://llvm.org/docs/LangRef.html, 2023.

[2] P. Alves, F. Gruber, J. Doerfert, A. Lamprineas, T. Grosser, F. Rastello, and F. M. Q. a. Pereira, "Runtime pointer disambiguation," in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[3] D. Callahan, J. J. Dongarra, and D. Levine, "Vectorizing compilers: A test suite and results," in *ACM/IEEE Conference on Supercomputing*, 1988.

[4] Y. Chen, C. Mendis, and S. Amarasinghe, "All you need is superword-level parallelism: Systematic control-flow vectorization with SLP," in *International Conference on Programming Language Design and Implementation*, 2022.

[5] K. Chitre, P. Kedia, and R. Purandare, "The road not taken: Exploring alias analysis based optimizations missed by the compiler," *Proceedings of the ACM on Programming Languages*, 2022.

[6] K. Chitre, P. Kedia, and R. Purandare, "Rapid: Region-based pointer disambiguation," *Proceedings of the ACM on Programming Languages*, 2023.

[7] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *International Conference on Programming Language Design and Implementation*, 2000.

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in

International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, 2004.

[9] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan, "A compiler framework for speculative analysis and optimizations," in *International Conference on Programming Language Design and Implementation*, 2003.

[10] B. Liu, A. Laird, W. H. Tsang, B. Mahjour, and M. M. Dehnavi, "Combining run-time checks and compile-time analysis to improve control flow auto-vectorization," in *International Conference on Parallel Architectures and Compilation Techniques*, 2023.

[11] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir, "A compiler framework for extracting superword level parallelism," in *International Conference on Programming Language Design and Implementation*, 2012.

[12] V. Porpodas and T. M. Jones, "Throttling automatic vectorization: When less is more," in *International Conference on Parallel Architecture and Compilation*, 2015.

[13] V. Porpodas, A. Magni, and T. M. Jones, "PSLP: Padded SLP automatic vectorization," in *International Symposium on Code Generation and Optimization*, 2015.

[14] V. Porpodas, R. C. O. Rocha, E. Brevnov, L. F. W. Góes, and T. Mattson, "Super-Node SLP: Optimized vectorization for code sequences containing operators and their inverse elements," in *International Symposium on Code Generation and Optimization*, 2019.

[15] V. Porpodas, R. C. Rocha, and L. F. Góes, "VW-SLP: auto-vectorization with adaptive vector width," in *International Conference on Parallel Architectures and Compilation Techniques*, 2018.

[16] L.-N. Pouchet, "PolyBench/C: the polyhedral benchmark suite," https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, 2021.

[17] M. H. Sujon, R. C. Whaley, and Q. Yi, "Vectorization past dependent branches through speculation," in *International Conference on Parallel Architectures and Compilation Techniques*, 2013.