Vector-DDG (Vector Data Dependence Graph) for Better Visualization and Verification of Vectorized LLVM-IR

Sumukh J Bharadwaj, Aloor Raghesh, Roomman Israili,
Karthik Kotikalapudi, Ayushman Singh, Meena Jain

AMD Compilers Team

together we advance_

Agenda

- INTRODUCTION
- VECTOR-DDG: VISUALIZER
- VECTOR-DDG: VERIFIER
- SOUNDNESS AND CORRECTNESS
- FUTURE WORK
- COPYRIGHTS

Introduction: Data Dependence Graph and Vectorization

- Vectorization brings big performance uplifts for various applications

- Presence of complicated data flow makes it difficult to comprehend the vectorized LLVM IR

- We propose Vector-DDG to address the above challenge
  - Extension to the state-of-the art Data Dependence Graph (DDG)
  - Visualization - helps developing insights to improve the quality of the vector code
  - Verification - to establish the correctness of the vector code
  - This verifier can also be used as a pass which scalarizes the code vectorized by SLP vectorizers

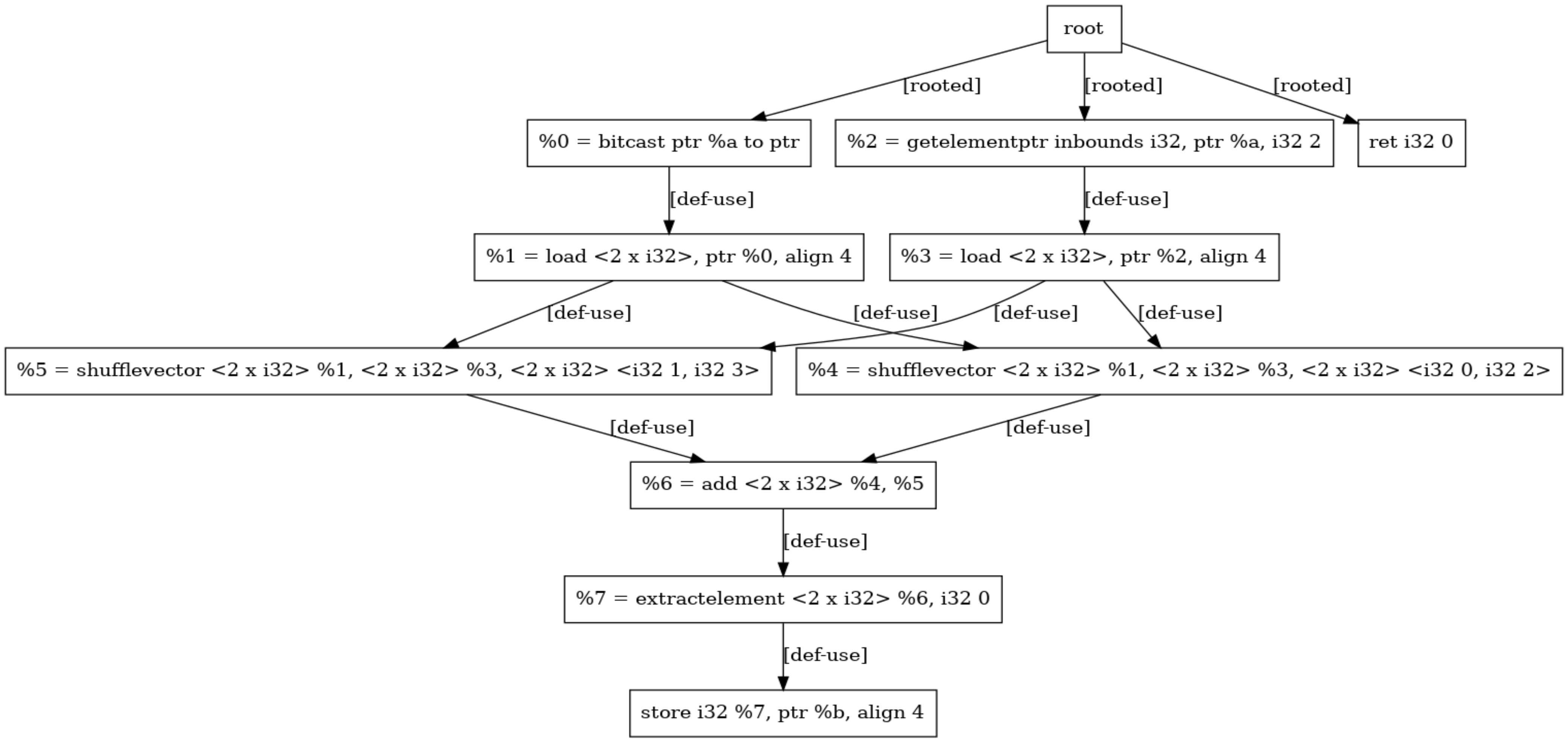
VectorDDG: Visualizer – An example

- The LLVM IR of the example being considered

```
define i32 @hadd(i32* %a, i32* %b) {
entry:
  %0 = bitcast ptr %a to ptr
  %1 = load <2 x i32>, ptr %0, align 4
  %2 = getelementptr inbounds i32, ptr %a, i32 2
  %3 = load <2 x i32>, ptr %2, align 4
  %4 = shufflevector <2 x i32> %1, <2 x i32> %3, <2 x i32> <i32 0, i32 2>
  %5 = shufflevector <2 x i32> %1, <2 x i32> %3, <2 x i32> <i32 1, i32 3>
  %6 = add <2 x i32> %4, %5
  %7 = extractelement <2 x i32> %6, i32 0
  store i32 %7, ptr %b, align 4
  %8 = getelementptr i32, ptr %b, i32 1
  %9 = bitcast ptr %8 to ptr
  store <2 x i32> %6, ptr %9, align 4
  ret i32 0
}
```

VectorDDG: Visualizer – An example

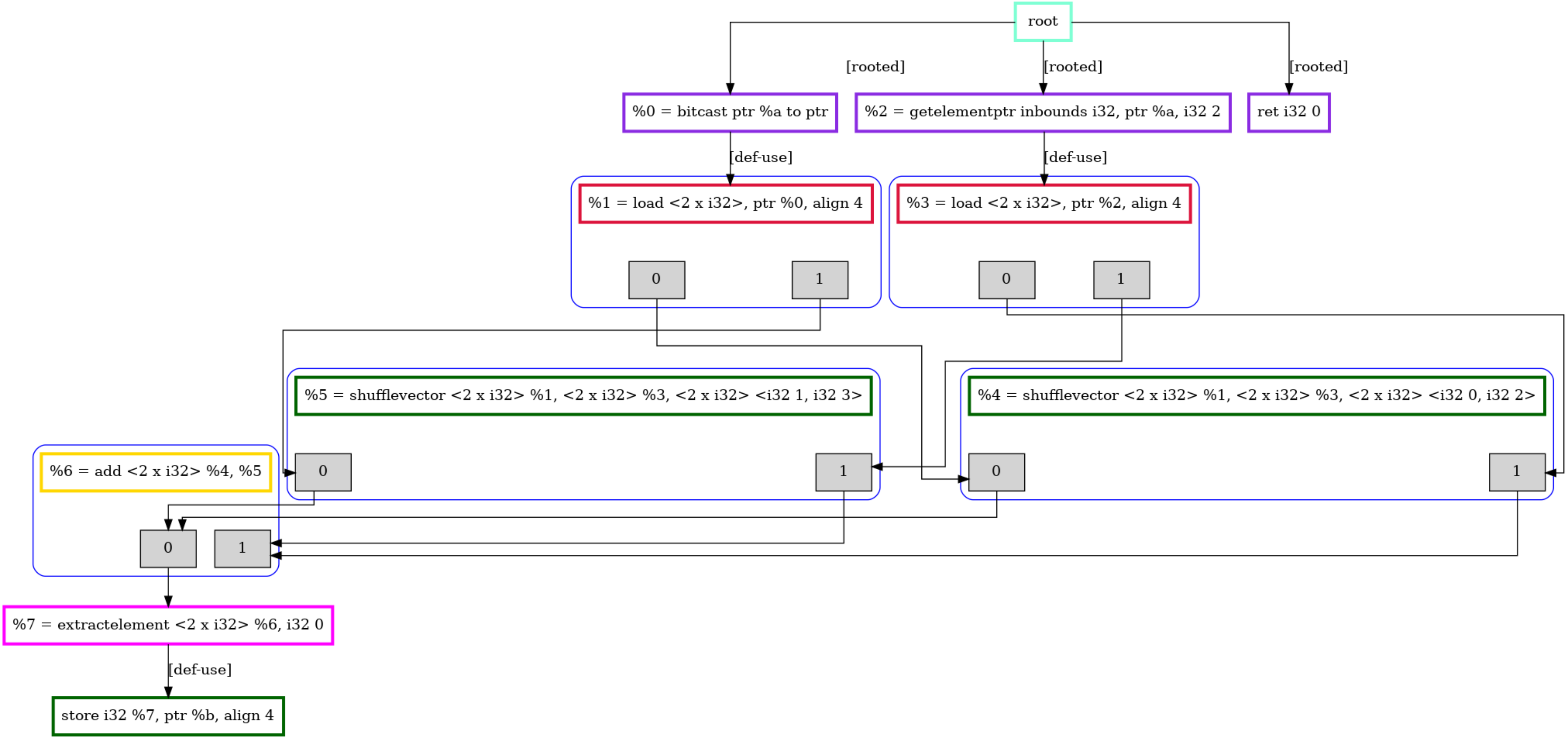
- Dependence graph without VectorDDG



DDG for 'hadd'

VectorDDG: Visualizer – An example

- Dependence graph with VectorDDG



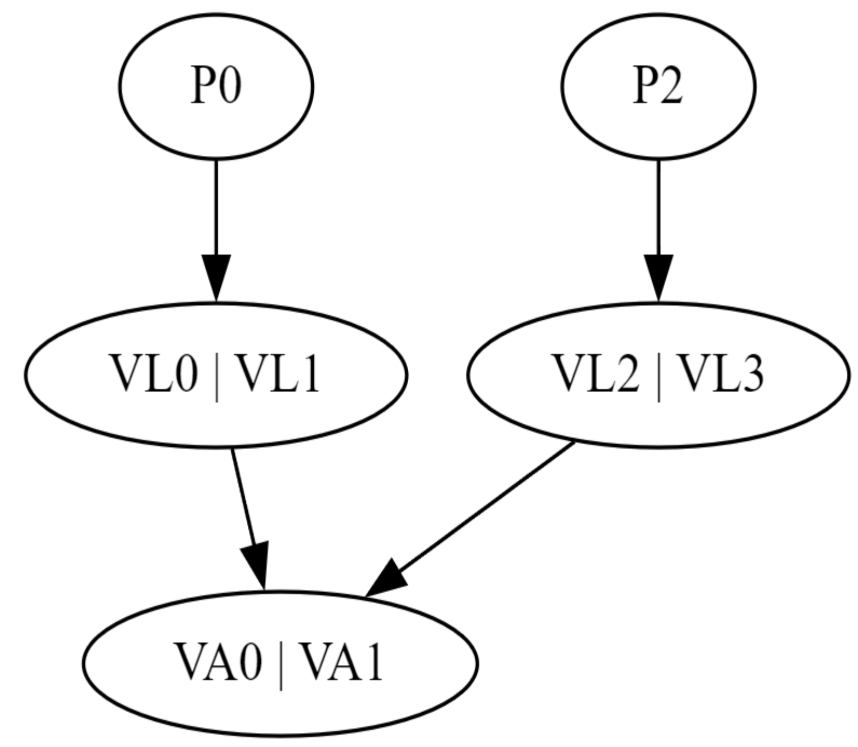
```
%0 = bitcast ptr %a to ptr
%2 = getelementptr inbounds i32, ptr %a, i32 2
ret i32 0
%1 = load <2 x i32>, ptr %0, align 4
%3 = load <2 x i32>, ptr %2, align 4
%5 = shufflevector <2 x i32> %1, <2 x i32> %3, <2 x i32> <i32 1, i32 3>
%4 = shufflevector <2 x i32> %1, <2 x i32> %3, <2 x i32> <i32 0, i32 2>
%6 = add <2 x i32> %4, %5
%7 = extractelement <2 x i32> %6, i32 0
store i32 %7, ptr %b, align 4
```

DDG for 'hadd'

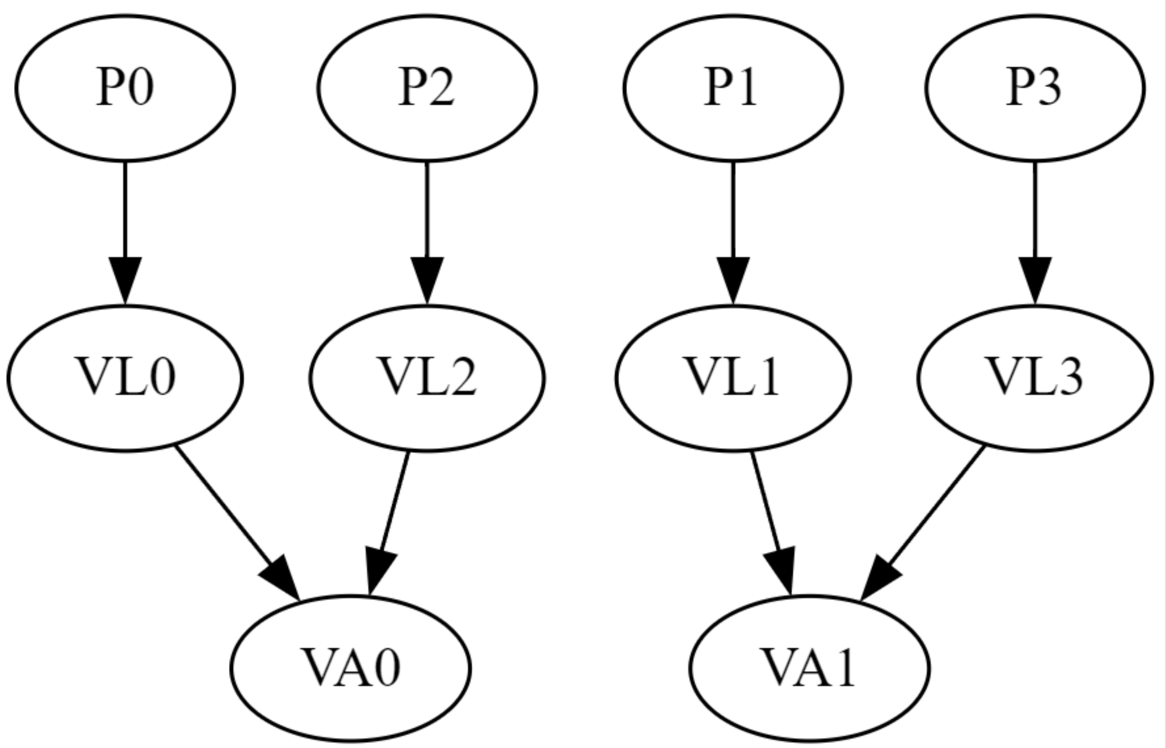
VectorDDG: Verifier – Scalarizing the Vector Lanes

- Split a vector node into multiple nodes corresponding to each lane

- Equivalent to scalarization of the Vector DDG



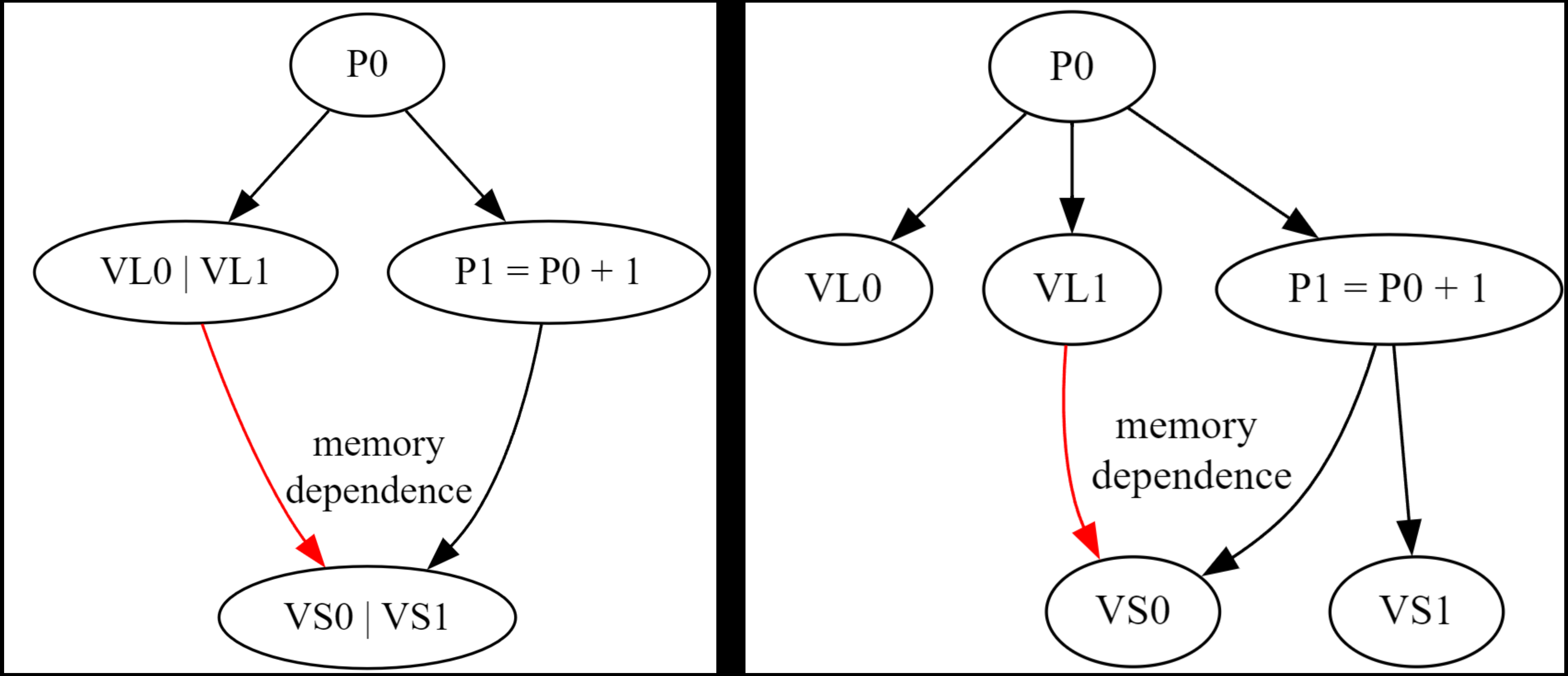
Original VectorDDG



Scalarized DDG with nodes for each lane

VectorDDG: Verifier - Scalarizing the Vector Lanes

- Precise memory dependence edges are added correspondingly



Indication of precise memory dependence

VectorDDG: Verifier

- Nodes: scalar instructions or vector lanes
- Edges: data or memory dependence edges
- Checks if there are same data and memory dependence in ScalarDDG and Scalarized VectorDDG
- Assumptions
  - Instruction set of the ScalarDDG is the same as the scalarized VectorDDG
  - SLP Vectorizer does not perform any non-trivial transformations

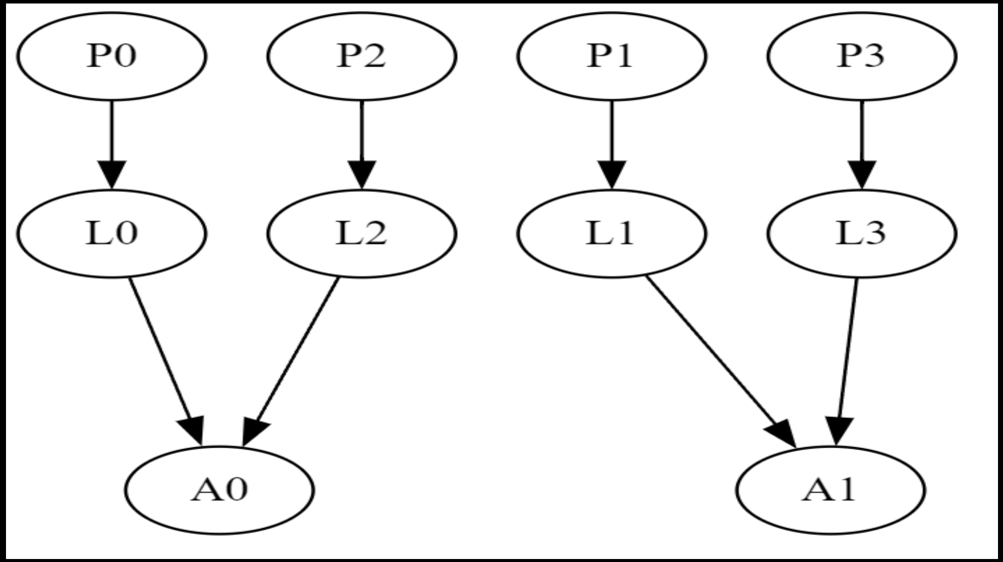
A Vector-DDG is equivalent to a Scalar-DDG if and only if for each path in Scalar-DDG there exists a unique path in the Scalarized Vector-DDG

VectorDDG: Verifier - Algorithm

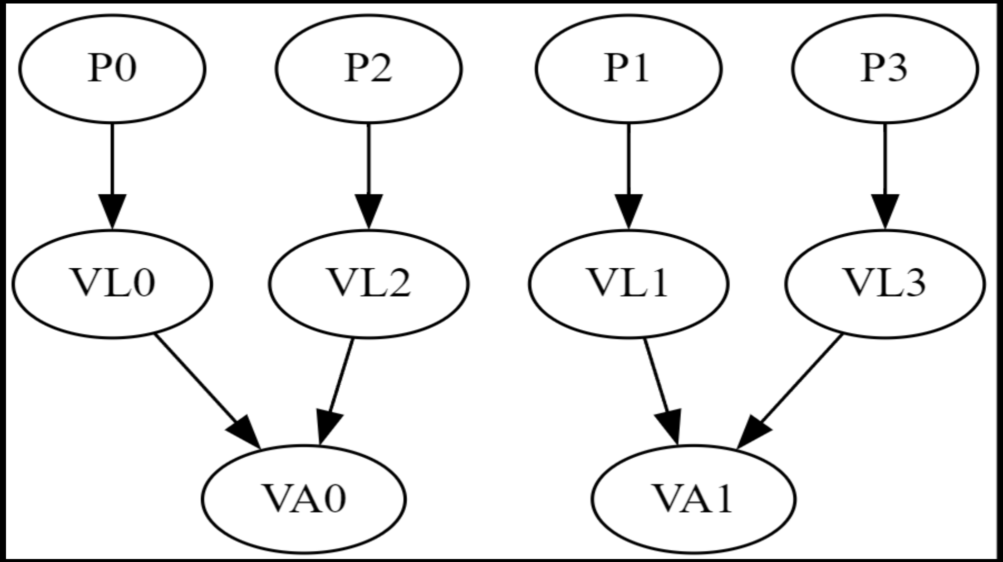
- We traverse both DDGs in topological order and perform a level-by-level comparison
- We first compare the external values (parameters, constants) of both the DDGs
- For the corresponding levels, we try to match the nodes by comparing their parents
- If the match fails at any point, we conclude that the DDGs are non-equivalent

VectorDDG: Verifier - Example

```
define void fscalar(ptr %P0, ptr %P1, ptr %P2, ptr3) {
  %L0 = load i32, ptr %P0
  %L1 = load i32, ptr %P1
  %L2 = load i32, ptr %P2
  %L3 = load i32, ptr %P3
  %A0 = add i32 %L0, %L2
  %A1 = add i32 %L1, %L3
}

define void fvec(ptr %P0, ptr %P1, ptr %P2, ptr %P3) {
  %VL0 = load <2 x i32> ptr %P0
  %VL2 = load <2 x i32> ptr %P2
  %VA = add <2 x i32> %VL0, %VL2
}
```



ScalarDDG representation



VectorDDG representation

Soundness

- Soundness could be defined as
  - Equivalent → Actually equivalent
  - Not-equivalent → it can be actually equivalent or not-equivalent

- **The verification procedure is sound in nature**

- Currently, in unhandled cases or scenarios where it is difficult to judge, we return the result as non-equivalent
  - Therefore, the verifier may label equivalent programs as non-equivalent

Future Work

- Propose this as an RFC to llvm community
- Propose this as a Google Summer Of Code project to enable further development
- Implement techniques to have visualization of subgraphs for dense Vector-DDGs
- Extend the support to Loop Vectorizer
- Implementation: Work in progress

[Public]

Copyright and disclaimer

- ©2024 Advanced Micro Devices, Inc. All rights reserved.

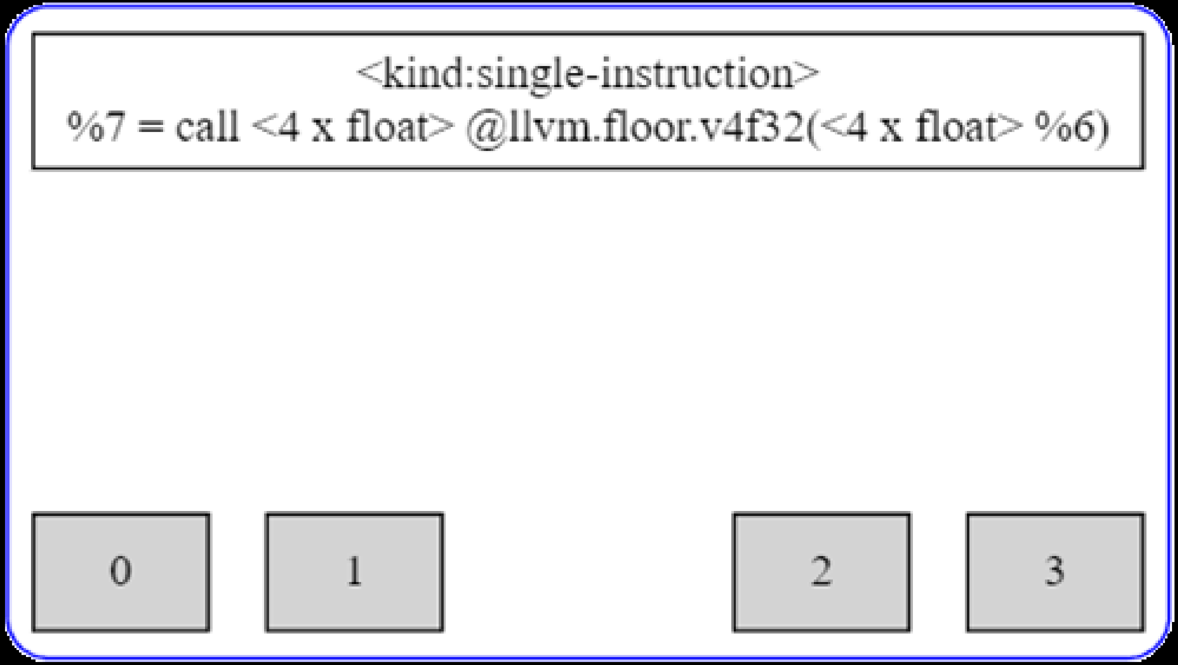
- AMD, the AMD Arrow logo, [insert all other AMD trademarks used in the material IN ALPHABETICAL ORDER here per AMD's Guidelines on Using Trademark Notice and Attribution] and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

- The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

- THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION

Appendix

VectorDDG: Visualizer – Node representation



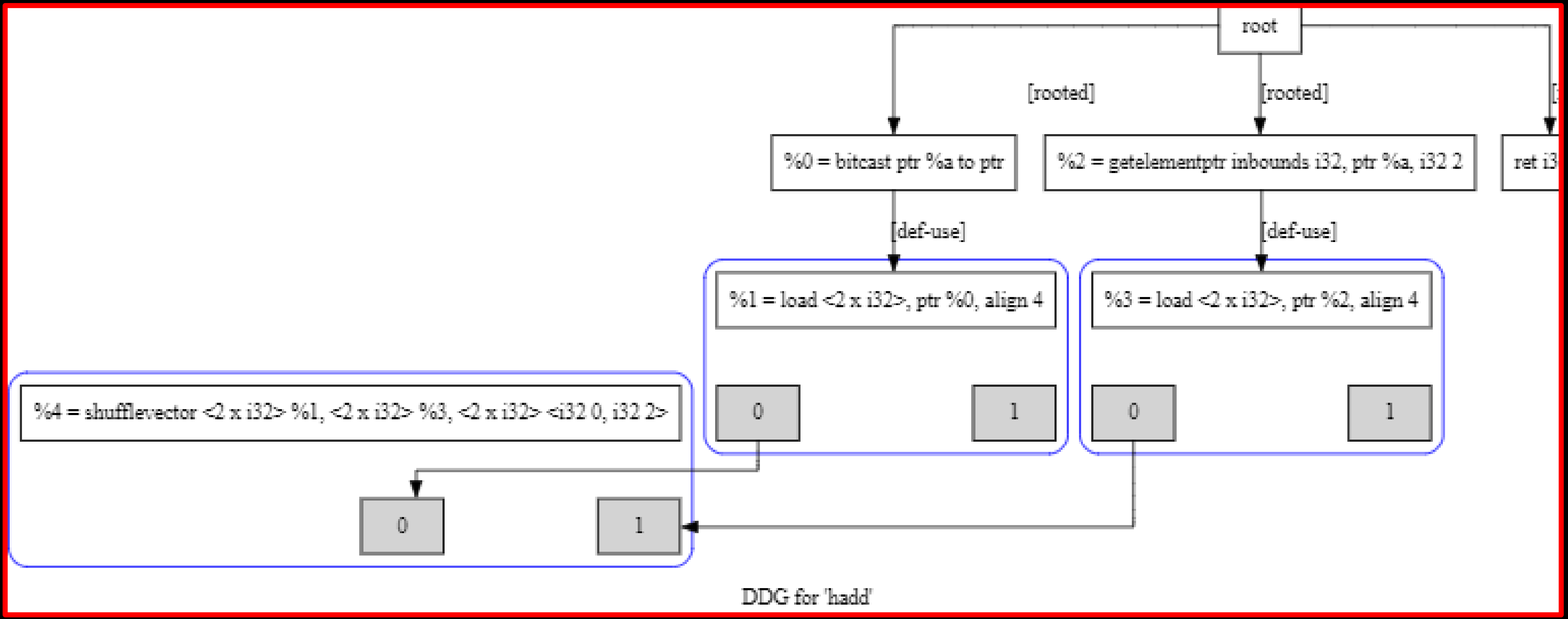
Node representation for vector instructions

- The nodes of the original graph bearing vector instructions are modified to represent the lanes associated with the vector being dealt with and will thus be recognized as an atomic node in itself.

- Connections to vector lanes are depicted as edges to the lane nodes created which helps us demonstrate the data dependence in vector instructions which would give us a clear picture of lane flow and movement

VectorDDG: Visualizer – Handling Special Instructions

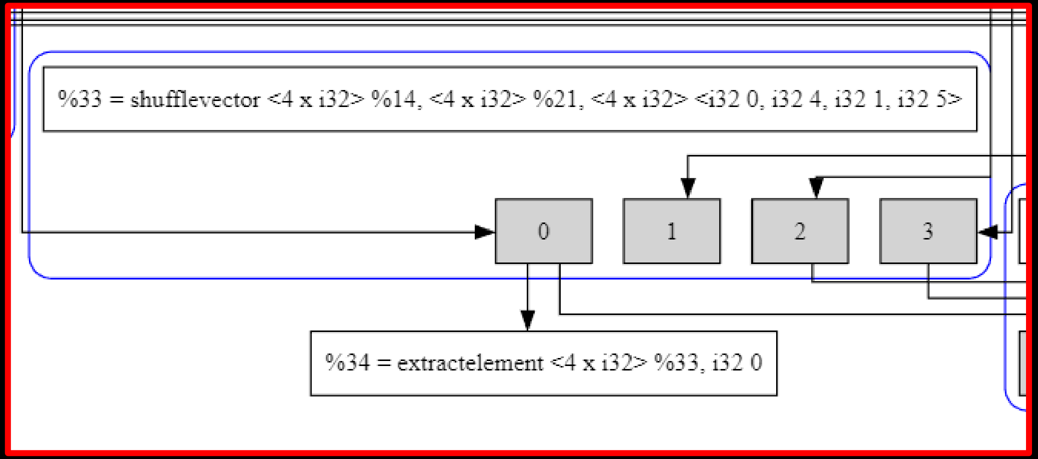
- **Shuffle vector instruction**: This instruction requires the mask input to be identified after obtaining the two vector operands in order to divert the lanes in a specified manner.
- The value of the index indicated by the mask must be extracted from the two vectors and the edges have to be drawn correspondingly.



Shufflevector representation

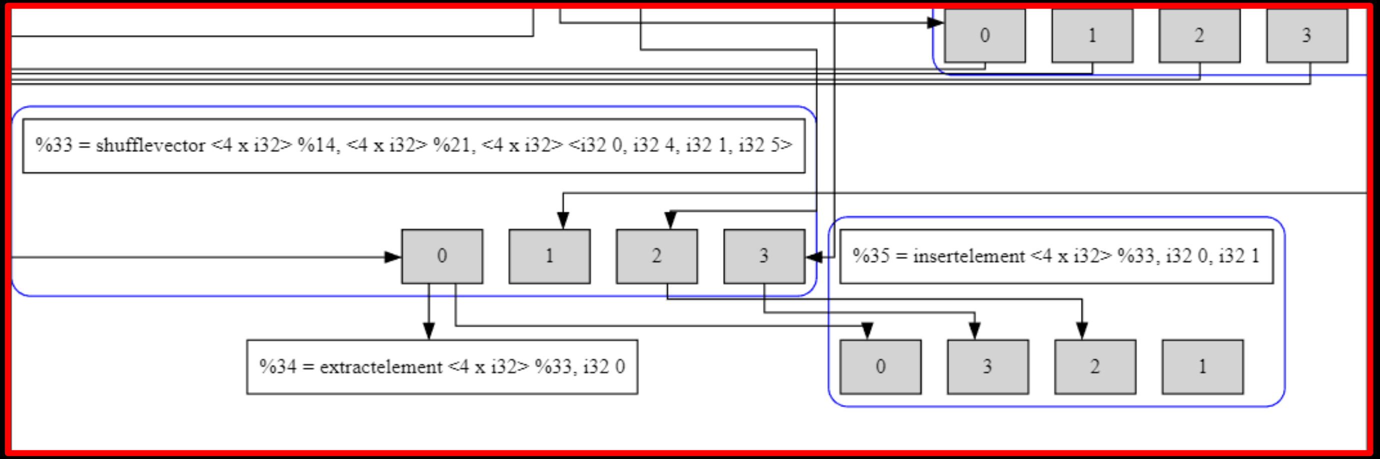
VectorDDG: Visualizer – Handling Special Instructions

- **Extract element instruction**: This instruction requires the index from the operand to be identified and the edge from the particular lane of the node needs to directed towards the scalar node.
- **Insert element instruction**: This instruction requires the index from the operand to be identified, thereby generating the remaining edges from the source node and the required element to be inserted from either a fixed scalar value or from the result of another operation



```
%33 = shufflevector <4 x i32> %14, <4 x i32> %21, <4 x i32> <i32 0, i32 4, i32 1, i32 5>
%34 = extractelement <4 x i32> %33, i32 0
```

Extract element representation



```
%33 = shufflevector <4 x i32> %14, <4 x i32> %21, <4 x i32> <i32 0, i32 4, i32 1, i32 5>
%35 = insertelement <4 x i32> %33, i32 0, i32 1
%34 = extractelement <4 x i32> %33, i32 0
```

Insert element representation

Problem with non-trivial transformations

- Assume that in the scalar code we have the following

```
%0 = add 10, 10
%1 = mul %0, 10
```

- These are not vectorizable instructions
- Assume that for some reason an earlier pass did not fold %0 into 20
- Consider the vectorized code where it was folded to 20

```
%1 = mul 20, 10
```

- This will be again folded to 200 and the above two instructions will be missing in the vectorized code
- This violates our notion of a one-to-one comparison for the same operation performed

VectorDDG: Verifier - Algorithm

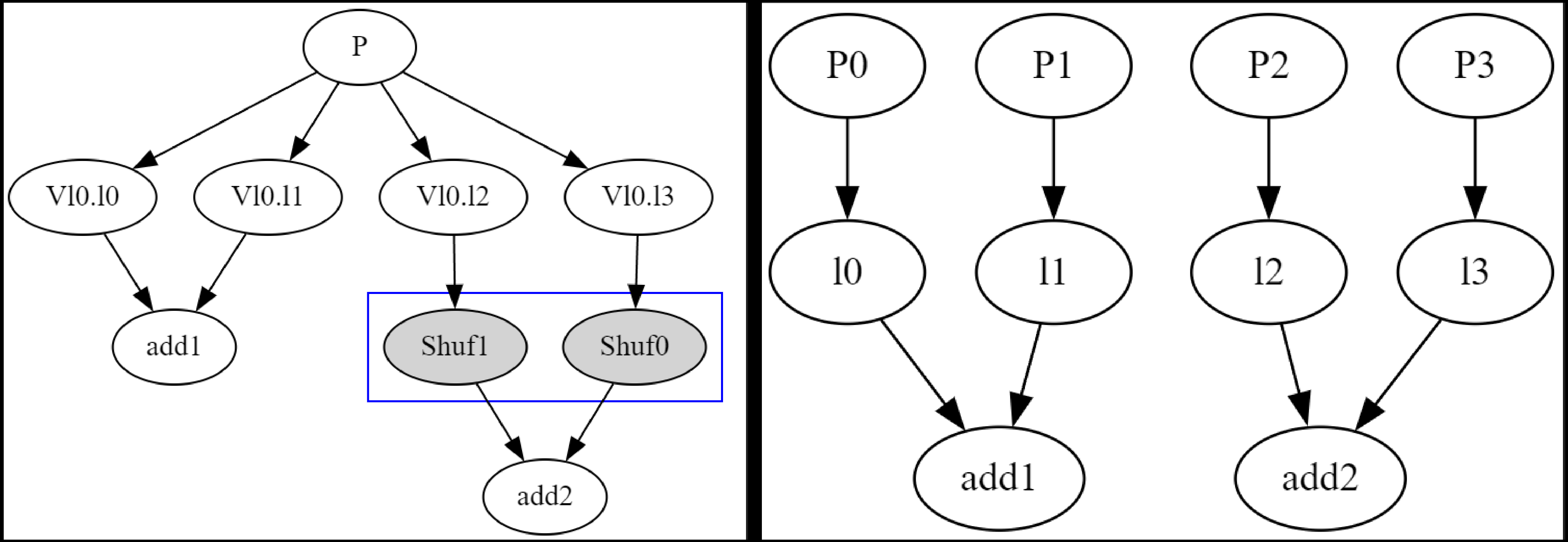
```
M1: ScalarNodeToIndegreeMap = findIndegreeForScalarDDG()
M2: VectorNodeToIndegreeMap = findIndegreeForVectorDDG()

CurScalarFrontier = {Scalar nodes with indegree of 0}
NextScalarFrontier = {}
CurVectorFrontier = {Vector nodes with indegree of 0}
NextVectorFrontier = {}
while(!CurScalarFrontier.Empty() && !CurVectorFrontier.isEmpty()){
    if (!match(CurScalarFrontier, NextScalarFrontier, CurVectorFrontier, NextVectorFrontier))
        return false
    CurScalarFrontier = NextScalarFrontier
    NextScalarFrontier = {}
    CurVectorFrontier = NextVectorFrontier
    NextVectorFrontier = {}
}
If (CurScalarFrontier.isEmpty() && CurVectorFrontier.isEmpty())
    return true
return false
```

VectorDDG: Verifier - Algorithm

```
match (CurScalarFrontier, NextScalarFrontier, CurVectorFrontier, NextVectorFrontier){
    bypassSpecials(CurScalarFrontier, M1)
    bypassSpecials(CurVectorFrontier, M2)
    for VectorNode in CurVectorFrontier:
        for ScalarNode in CurScalarFrontier:
            if ( sameParents(VectorNode, ScalarNode) ) {
                for each child of VectorNode:
                    M2[child] --
                    if (M2[child] == 0)
                        NextVectorFrontier.insert(child)
                for each child of ScalarNode:
                    M1[child] --
                    if (M1[child] == 0)
                        NextScalarFrontier.insert(child)
                break
            }
    If (CurScalarFrontier.isEmpty() && CurVectorFrontier.isEmpty())
        return true
    return false
}
```

VectorDDG: Verifier – Handling Special instructions

- Special instructions that modify the lane ordering or bridge scalar and vector instructions can be mapped by taking only the flow in consideration while bypassing the instructions
- This includes the ShuffleVector, InsertElement and extractelement instructions



Shuffle bypass and rewire

VectorDDG: Verifier – Preprocessing GEPs

- GEPs are special instructions that have no direct correspondence with the scalar counterpart, nor can they be ignored and bypassed due to their nature

- GEPs are preprocessed by modifications to scalar components and verifying the equivalence in the process

- For each corresponding lane node in a vector load we find the matching GEPs in this scenario
  - a) If the particular GEP has other uses then duplicate the GEP and update the parents of the use
  - b) If the GEPs parent is another GEP, merge the parent recursively while duplicating it if other uses are found

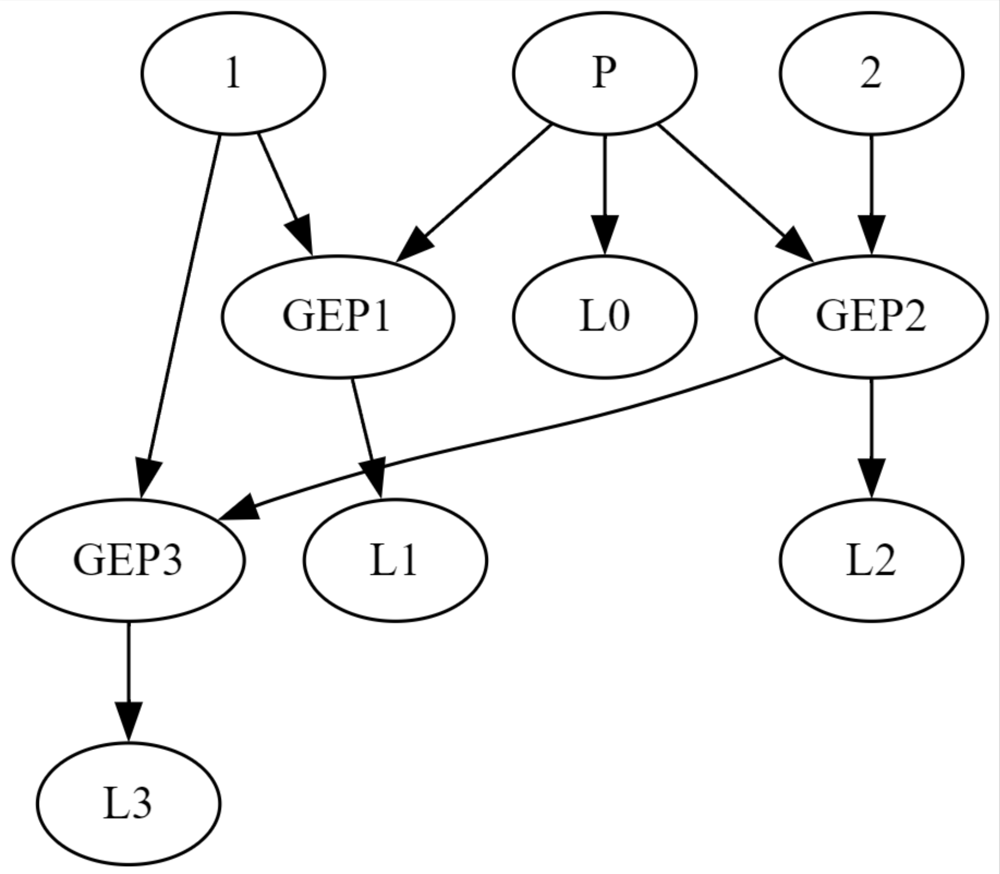
VectorDDG: Verifier – Preprocessing GEPs

- In the merged nodes of ScalarDDG , we store the corresponding vector lane to make the matching simpler
- This process also performs partial verification by making rejections stating the nonequivalence of the vectorization when the comparison fails midway
- Assumption: This is a maximum of one store and one load for the GEP instruction being matched

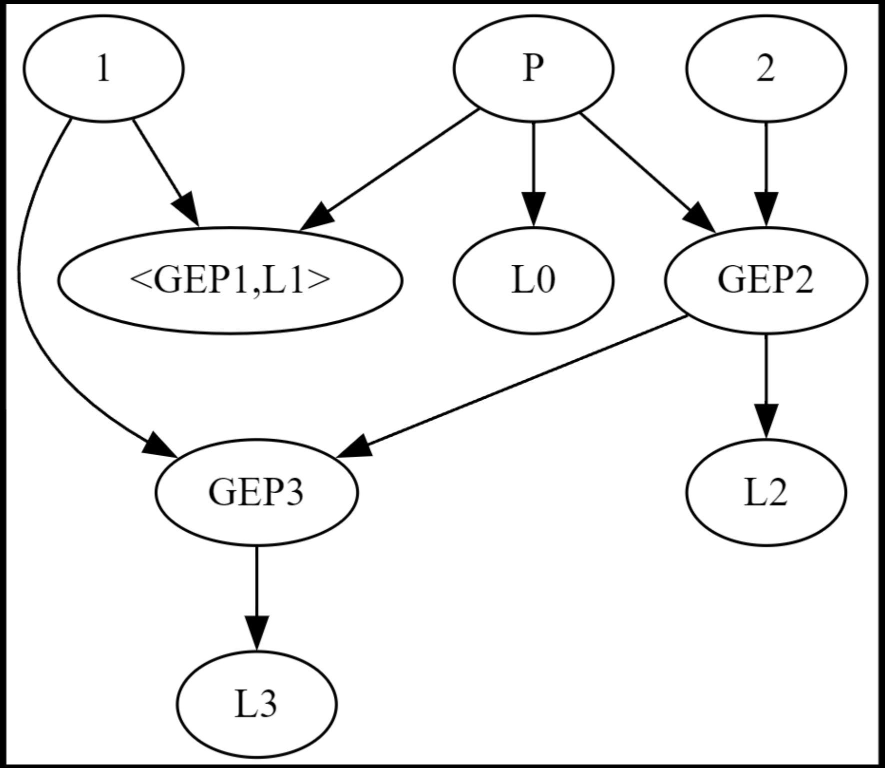
VectorDDG: Verifier – Preprocessing GEPs Example

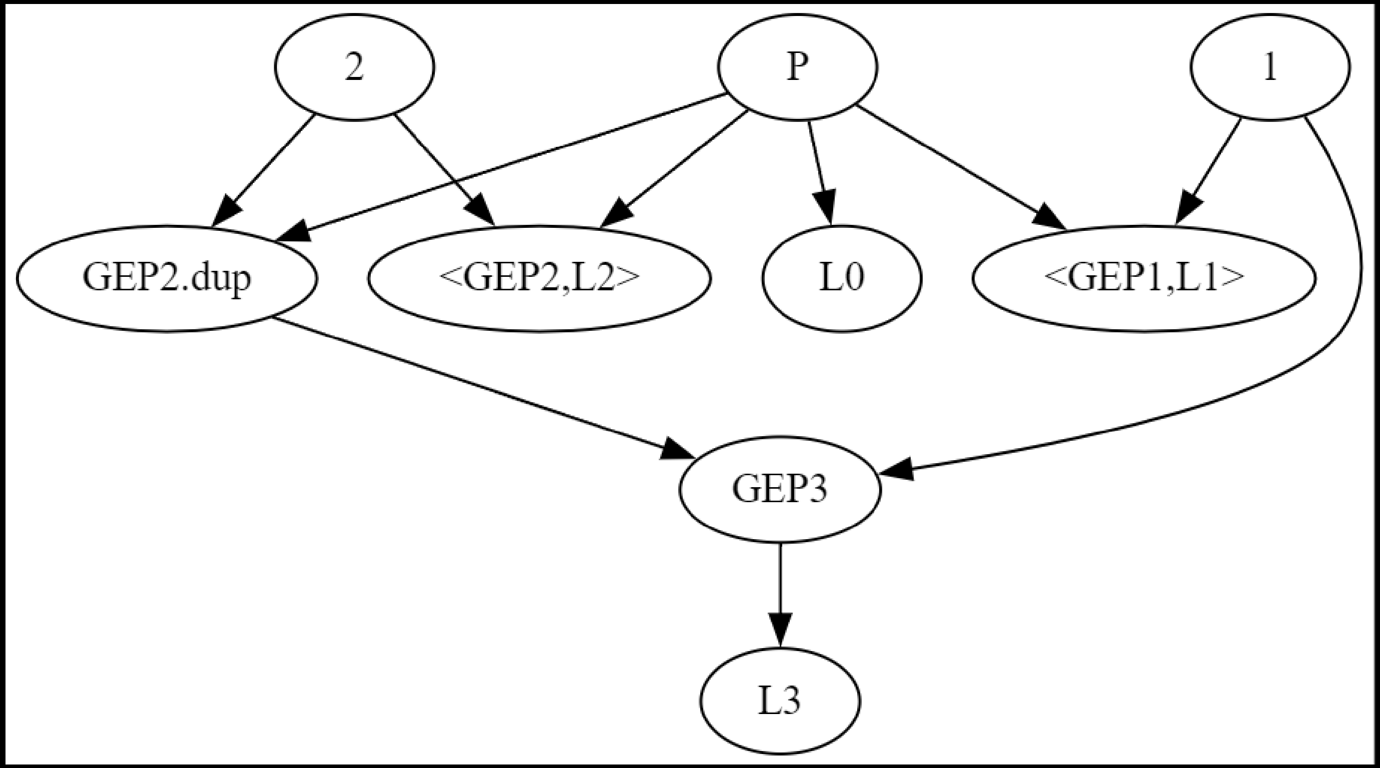
ScalarDDG with IR:

```
define void @gep(ptr %P) {
entry:
  %L0 = load i32, ptr %P
  %GEP1 = getelementptr i32, ptr %P, i32 1
  %L1 = load i32, ptr %GEP1
  %GEP2 = getelementptr i32, ptr %P, i32 2
  %L2 = load i32, ptr %GEP2
  %GEP3 = getelementptr i32, ptr %GEP2, i32 1
  %L2 = load i32, ptr %GEP3
  ret void
}
```



VectorDDG: Verifier – Preprocessing GEPs Example

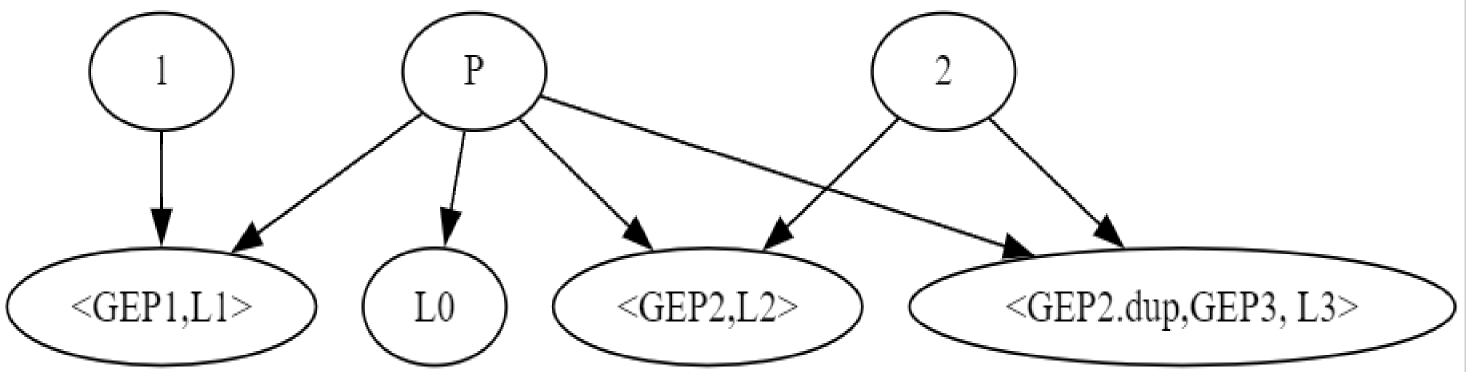




GEP handling and merging nodes for comparison

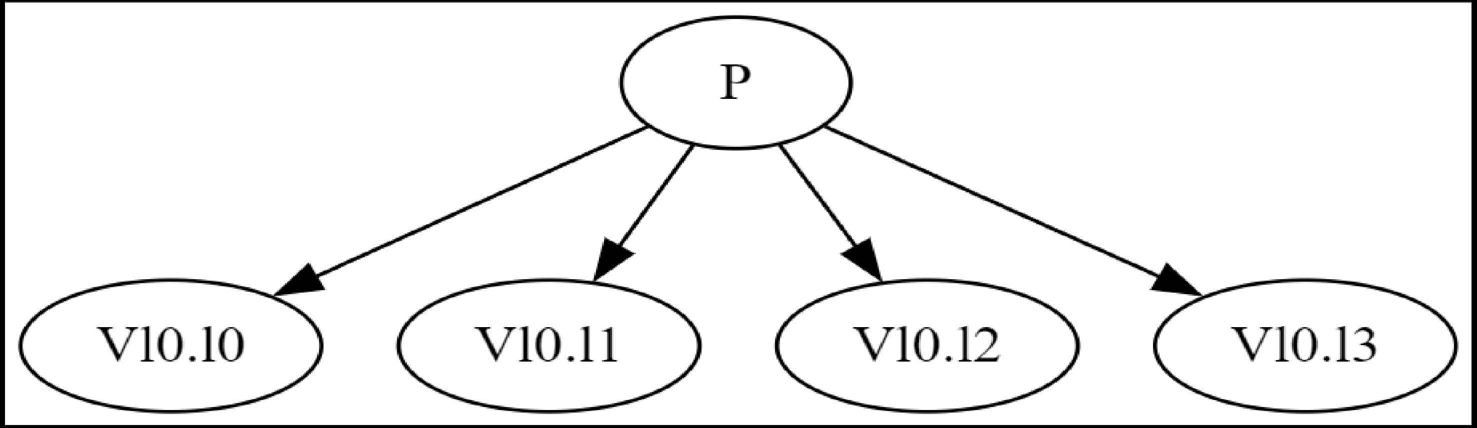
together we advance_

VectorDDG: Verifier – Preprocessing GEPs Example



Final ScalarDDG after GEP preprocessing

```
define void @gep.vec(ptr %P){
entry:
  %0 = load <4 x i32>, ptr %P
}
```



Corresponding VectorDDG

Proof by induction

- In each step, The algorithm proceeds by trying to find a matching frontier in Scalar-DDG for a frontier in Vector-DDG
- Note that we define frontier in topological order, not the usual BFS frontier
- The proof is by induction where we assume that all the parent frontiers in topological order are matched
- Now to match a vector frontier to a scalar one, we need to find matching nodes between them
- A node in vector frontier is matching to a node in scalar frontier if we can find uniquely matching parents for them
- If we cannot find we say that they are not equivalent