Utilizing Parallel Workers: LLVM's Vectorization Plan

Jonas Fritsch
Technical University of Munich
jonas.fritsch@tum.de

Abstract

Modern SIMD processors provide various vector registers and ISAs for programmers to utilize. However, manually vectorizing code can be time-consuming, so compilers look to auto-vectorize code for better performance. LLVM, as a widely used compiler toolchain, could only auto-vectorize basic scalar code. Intel proposed a comprehensive refactoring of the underlying system to enable LLVM's auto-vectorization to support more complex optimizations. The suggested Vectorization Plan (VPlan) architecture would be more modular and scalable. Since then, significant efforts have been made to implement this new system. Today, VPlan is already actively involved in vectorizing innermost loops. Other areas, such as outer loop vectorization, are also in development.

1 Introduction

Modern CPUs are often equipped with multiple different vector registers. These registers are nowadays as wide as 512 bits, allowing for the processing of multiple data streams at once (SIMD). By batching multiple values together in one register, different ISAs like Intel AVX or ARM SVE allow the execution of the same instruction for all values in a vector register simultaneously. Utilizing this leads to significant performance improvements over the scalar equivalent most of the time.

However, manual code vectorization can quickly become time-consuming, especially when supporting different CPU architectures. Modern compilers aim to automatically transform scalar code to use vectorization when applicable.

As one of the most widely used compilation frameworks, LLVM [4] has implemented and refined its auto-vectorization over many years. It provides two different Vectorizers, one for innermost loops (LoopVectorize) and one for super-word parallelism (SLPVectorize) [14].

This system, however, had quite a few limitations, as the loop vectorizer could only handle innermost loops and neither outer loops, complex control flow, nor non-inlined functions. Additionally, while multiple different vectorizations for the same scalar code would be possible, the current vectorizers working directly on the LLVM IR could not model and compare the costs of such different vectorization approaches.

With these limitations in mind, Intel started an ongoing refactorization effort to migrate LLVM's auto-vectorization pipeline to utilize a more abstract Vectorization Plan (VPlan) [12, 15]. The final goal would be to unite LLVM's auto-vectorization in a single flexible system capable of optimizing SLPs, inner, and nested loops with complex control flows. The auto-vectorization pass would create and transform multiple different VPlans, each modeling a different vectorization approach, abstracted away from the underlying LLVM IR. Those VPlans are compared against each other based on a cost model, and finally, the best one, which might also not vectorize at all, is materialized into the LLVM IR.

LLVM's VPlan is currently being integrated into the existing Loop Vectorizer and is already modeling most inner loop vectorizations and transformations [3]. Vectorization for outer loops is also in development and can be enabled by setting the `-enable-vplan-native-path` feature flag [1, 2]. In the future, the plan will be to merge both of these loop vectorization paths into one. Merging the existing SLP Vectorizer with VPlan is also planned.

2 Background

In general, vectorization techniques can be separated into different categories:

2.1 Inner Loop Vectorization

Innermost loops are loops whose body's control flow does not contain any more loops, bound or unbound. Vectorization of these loops is more straightforward

```
1 for (int i = 0; i < N; i++) {
2     if (z[i]) {
3         x[i] += y[i];
4     }
5 }
```

Figure 1. A code snippet of a simple loop that would be vectorized over the variable `i`. The conditional branch would be flattened into a masked addition. The mask would be generated based on `z[i]`.

as the only option to vectorize is over the loop's induction variable.

There are various ways to transform the control flow of a loop to accommodate operating on widened vector registers. Figure 1 shows an example of an if-conversion. Some more examples can also be found in LLVM's Auto-Vectorization Documentation [14].

2.2 Super-Word Parallelism (SLP) Vectorization

SLP Vectorization vectorizes similar independent instructions. A compiler might have, for example, fully unrolled a smaller loop in a previous code simplification pass. The resulting similar scalar instructions might then be vectorized later by an SLP vectorizer.

2.3 Outer Loop Vectorizaton

Outer loop vectorization focuses on control flow with nested loops. Such loops often present more challenges for auto-vectorization, as one could vectorize over the outer loop, inner loop, or a mix of both, representing a hybrid/multi-dimensional vectorization approach. Carrying out and comparing costs of such different vectorization approaches can be complex and require a flexible system like, for example, VPlans.

Figure 2 shows a nested loop code snippet where vectorization over the outer loop would be beneficial.

2.4 Function Vectorization

Function vectorization refers to the vectorization of entire functions. A loop might contain a non-inlined function call, passing data related to the iteration variable. During auto-vectorization, this call could

```
1 for (int i = 0; i < N; i++) {
2     float sum = 0;
3
4     for (int j = 0; j < 6; j++) {
5         sum += y[j][i] * z[j];
6     }
7
8     x[i] = sum;
9 }
```

Figure 2. A code snippet of a nested loop where it would be beneficial to vectorize over the outer loop. The inner loop has a trip count that is too low (6) to vectorize. Additionally, the memory access pattern of `y` would be consecutive compared to scattered when vectorizing over `i` instead of `j`.

be replaced with a call to a newly constructed function with the same semantic control flow but operating on multiple values at once through vector registers instead.

2.5 Vectorization Similarities

All of the above-mentioned vectorization techniques can be brought into relation with each other. For example, function vectorization can be transformed into loop vectorization by creating a loop around the function body and vectorizing it and vice versa. The same applies for SLP Vectorization and loop vectorization when unrolling a loop.

2.6 Vectorization Legality

Not all loops can be vectorized. There are various factors that can ultimately hinder any vectorization attempts.

Some of the most common vectorization barriers are data dependencies. As the vectorized version of a code executes for Vectorization Factor (VF) N data streams simultaneously, it inherently changes the order of operations. To guarantee that the semantics of the code do not change, all operations on these N data streams must, generally, be independent. The simplest case where this is not provided is when the i-th loop iteration depends on any of the $(i-N)$-th previous iterations.

While explicit data dependencies are easier to see, a simple loop, as in Figure 1, could carry an implicit dependency in line 3 if the `x` and `y` point to overlapping memory regions (pointer aliasing).

Not all data dependencies directly hinder any vectorization. One exception can sometimes be made for reduction idioms, as seen in the inner loop in Figure 2, where `sum` would be a reduction variable.

Additionally, the control flow should be primarily straight-line code. While most if-else constructs can be flattened to always execute both branches along with masking, the same is not valid for switch-case statements or too complex control flows.

2.7 Vectorization Costs

While vectorizing a given control flow may not be legal, the vectorized code sometimes performs worse than its scalar equivalent.

Some control flow transformations that seem beneficial at compile time might reveal a performance bottleneck at runtime. Considering the code snippet from Figure 1, it might be that almost all `z[i]` evaluate to false at runtime. The branch predictor of a CPU would then be able to guess the correct branch in the scalar code. The vectorized version would always load the `y[i]` into memory only for the addition to be masked away. Different approaches to avoid such runtime penalties are branch-on-superword-condition code (BOSCC) [8, 13] or active-lane consolidation (ALC) [9, 10].

Some vector instructions can be costly on most CPUs. Examples are gather, scatter, shuffle, and permutation instructions. Control flow with many non-adjacent memory loads/stores or operations that combine different vector elements, such as reductions, can decrease vectorization performance.

Additionally, the vectorized code size will always be larger than the scalar equivalent. This is due to many factors: (1) The transformed control flow often includes more instructions. (2) As the loop trip count might not be a perfect multiple of the vectorization factor, a scalar epilogue/remainder of the loop might be used to handle the tail iterations. (3) Various runtime checks might be inserted throughout the vectorized control flow to handle, for example, pointers to overlapping memory regions, in which case the scalar version of the loop is executed.

3 LLVM's Vectorization Plan

3.1 History

LLVM implements two different vectorizers for auto-vectorization, LoopVectorize (LV) and SLPVectorize. These vectorizers are a part of LLVM's middle-end and run after module simplification has taken place. As VPlan still needs to be integrated into SLP-Vectorization, we will only focus on explaining the original LoopVectorize structure.

LoopVectorize was designed to focus only on vectorization of innermost loops [11]. When starting with an initial scalable loop in IR, it would first run a legality phase to check whether and how it is legal to vectorize the loop. This phase would produce artifacts/maps, such as if and what runtime aliasing checks would be required. Then, the cost model ran next, creating more artifacts based on the scalar IR and legality artifacts. These cost model artifacts would associate different vectorization factors with a cost. Lastly, the transformation phase would vectorize the underlying IR in one go based on all the produced artifacts.

However, this rigid flow of LoopVectorize presented a challenge. Extending LV to include outer loop and nested loop vectorizations would have been difficult, as LV was designed to flatten a loop's control flow into a single basic block. Producing and considering more and more artifacts when trying to transform the scalar loop into a vectorized one all at once also did not scale well.

As a potential solution, Intel proposed a long-term effort to refactor LLVM's whole auto-vectorization using a system called Vectorization Plan (VPlan) [12].

The goal was to unite any auto-vectorization systems (SLP, Inner-/Outer-Loop) into a single flexible system [11, 17], as seen in Figure 3. This system would use VPlans to represent different vectorization approaches abstracted away from the underlying IR. Each VPlan would calculate its cost and transform itself to modify the final IR. At first, the Legality phase (yellow) would check if the scalar code would be legal to vectorize at all, and if so, initial VPlans would be constructed. In the following Planning phase (magenta), these initial VPlans would be optimized using various VPlan-to-VPlan transformations [2]. The VPlan's cost model would supervise those transformations. Finally, the best VPlan would be chosen and materialized/executed, modifying the underlying IR.

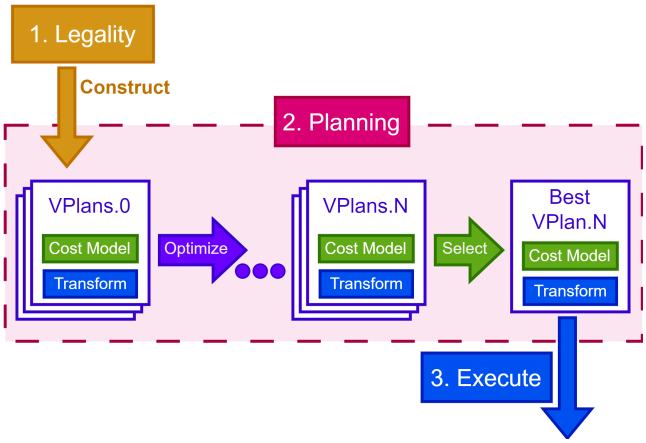

Figure 3. The planned future architecture for VPlan. The VPlan optimization would work as iteratively applying VPlan-to-VPlan transformations.

3.2 State of VPlan (LLVM 19.1.4)

In its current state, VPlan is deeply integrated into LV. When auto-vectorizing, LV starts by checking all the different loops in the IR. If it finds one, it will try to vectorize it (`LoopVectorizePass::processLoop`). Several factors might hinder vectorization for a given loop: (1) If the loop was already vectorized before, it will not be vectorized again. (2) The Legality phase checks if and how it is legal to vectorize the scalar loop. (3) The loop's trip count might be too small to make vectorization profitable. (4) The cost of runtime checks, such as pointer aliasing, can be too high.

The initial VPlans are constructed based on the legality checks and the legacy LV cost model's cost decisions referencing the scalar IR (`LoopVectorizationPlanner::plan`). After initial VPlan construction, various VPlan-to-VPlan transformations are applied to optimize it (`LoopVectorizationPlanner::buildVPlansWithVPRecipes`).

Then, the optimal vectorization factor (VF) is calculated based on the set of VPlans and their representative cost (`LoopVectorizationPlanner::cost`), the calculation of which still involves the legacy LV cost model. The best VPlan is picked based on the optimal VF. A VF of 1 means it is not profitable to vectorize the loop; however, interleaving/unrolling can still modify the underlying IR. Afterwards, the Interleave Count (IC) is determined (`LoopVectorizationCostModel::selectInterleaveCount`).

```
for.body:
  %indvars.iv = phi i64 [ 0, %for.body.preheader ],
                        [ %indvars.iv.next, %for.inc ]
  %arrayidx = getelementptr inbounds i8, ptr %z, i64 %indvars.iv
  %0 = load i8, ptr %arrayidx, align 1
  %tobool.not = icmp eq i8 %0, 0
  br i1 %tobool.not, label %for.inc, label %if.then

if.then:
  %arrayidx2 = getelementptr inbounds float, ptr %y, i64 %indvars.iv
  %1 = load float, ptr %arrayidx2, align 4
  %arrayidx4 = getelementptr inbounds float, ptr %x, i64 %indvars.iv
  %2 = load float, ptr %arrayidx4, align 4
  %add = fadd float %1, %2
  store float %add, ptr %arrayidx4, align 4
  br label %for.inc

for.inc:
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond.not = icmp eq i64 %indvars.iv.next, %wide.trip.count
  br i1 %exitcond.not, label %for.cond.cleanup, label %for.body
```

Figure 4. The scalar IR for the main loop section of the conditional loop depicted in Figure 1. Note that `getelementptr inbounds` was abbreviated to `getelmptr inbs`.

The best VPlan is then executed (`LoopVectorizationPlanner::executePlan`). In the case of vectorization (VF $>$ 1), LV will also directly try to vectorize the epilogue loop.

3.2.1 VPlan Structure

To better understand the structure of a VPlan, we will use the simple conditional loop from Figure 1 as an example.

Figure 4 shows the scalar IR of the main loop basic blocks `for.body`, `if.then`, and `for.inc`. Figure 5 shows the equivalent hierarchical control-flow-graph (CFG) of a VPlan modeling a vectorization approach for this scalar loop. Other basic blocks like the scalar epilogue loop and its predication check (is the final vector trip count equal to the original trip count) are omitted. See Figure 6 in Appendix A for the fully modeled HCFG.

The different colored parts in the scalar IR map to the modeled VPlan region with the same color. The only new part that VPlan models in this example is handling the scalar epilogue loop (purple).

The hierarchical CFG of a VPlan helps abstract it away from the underlying IR. The nodes of this CFG are either `VPBasicBlocks`, containing a sequence of one or more `VPRecipes`, or `VPRegionBlocks`, modeling a single-entry-single-exit subgraph. In the example, `vector loop` is a region block. All other blocks are basic blocks.

`VPRegionBlocks` can indicate that their contents are to be replicated a certain number of times to model more complex control flow for different VF-UF combinations more efficiently in a single VPlan.

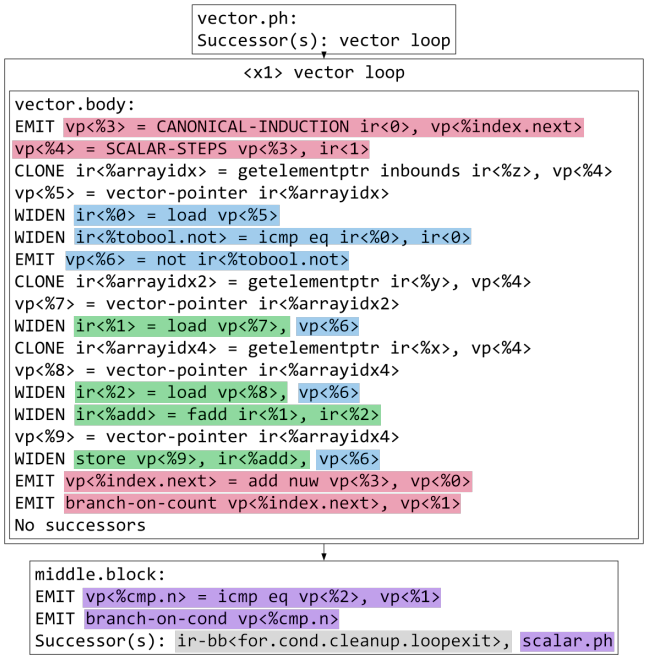

Figure 5. The vectorized loop section of the hierarchical CFG of a VPlan modeling the conditional loop depicted in Figure 1. Some predefined values are `vp<%0>` = VF * UF; `vp<%1>` = vector-trip-count; `vp<%2>` = original trip-count.

In Figure 5, `vector loop` is not to be replicated (`<x1>`).

Within our two `VPBasicBlocks` are lists of recipes (lines of instructions). A recipe models one or more instructions in the executed IR and can be based on one or more instructions from the input IR (ingredients). Note that a recipe might also model an instruction with no reference to any input IR (in our example, this would be the inversion of our vector mask). There are different types of recipes. `VPReplicateRecipe` will either `CLONE` exactly one instruction from the input IR into exactly one instruction in the output IR, or `REPLICATE` an instruction a certain number of times into output IR (typically found in replicator `VPRegionBlocks`). Different `VPWidenRecipes` exist to transform/`WIDEN` any input IR instruction into a vectorized version of it. The last remaining recipe type in our example is `VPInstruction`. This recipe `EMIT`s new instructions into the output IR that have no representative in the input IR (no ingredients).

Furthermore, VPlan recipes can also model definitions and usages of different values. In our example, the comparison mask `vp<%6>` is a `VPValue` used by three `VPUsers`.

3.3 Future of VPlan

Many of the VPlan internals and surrounding systems are still in active development. To stay up-to-date with the latest developments regarding auto-vectorization in LLVM, one can follow the Community Forum[1] or attend the monthly online meetings[2].

3.3.1 VPlan and Integration with LV

Setting up and integrating the VPlan structure into LV has seen tremendous effort over the last few years. Florian Hahn presented future directions for VPlan [3], which mostly revolve around transforming more LV-based decisions into explicit VPlan-to-VPlan transformations. Additionally, the legacy LV cost model still influences many cost model-related decisions. However, the transition to a VPlan-based cost model has already started.

3.3.2 Outer Loop Vectorization

Development around outer loop vectorization has slowed as the focus shifted to VPlan internals and integration into LV. The inner and outer loop paths are still separated, and the experimental `-enable-vplan-native-path` flag must be set to activate any outer loop vectorization. Further development of the outer loop path and eventually merging it with the innermost loop path is planned.

3.3.3 Merging with SLP Vectorize

Merging the existing SLP Vectorizer with the current VPlan infrastructure is in the early planning phase.

3.3.4 Function Vectorization

There is currently no active development for whole-function vectorization in the llvm-project's upstream.

4 Related Work

Next to the main llvm-projects upstream, many projects are using or extending the LLVM toolchain.

[1] https://discourse.llvm.org/c/ir-optimizations/loop-optimizations/62
[2] https://discourse.llvm.org/t/monthly-vectorizer-online-sync-up/78978/3

One such project that also tries to enable auto-vectorization support for inner-, outer-loop, and whole-function vectorization is the region vectorizer from the University Saarland [16]. This region-vectorizer implements partial control-flow linearization [6] and a divergence analysis that can handle unstructured CFGs. There has been an effort to partially implement this control-flow linearization and divergence analysis into LLVM's upstream [5, 7].

5 Summary and Future Work

In conclusion, VPlan has been in active development for several years now. When compiling with LLVM, VPlans model initial vectorization approaches, keep track of different optimization decisions, and finally, the best VPlan executes to transform the underlying IR. However, some VPlan-to-VPlan transformations and a VPlan-based cost model must be included before the legacy LV code can be discarded entirely.

Outer loop vectorization support via VPlan is still experimental, separated from the inner loop LV integration, and runs on a divergent experimental path (`-enable-vplan-native-path`). In the near future, development on outer loop vectorization will continue, and merging it with LV's VPlan path is planned.

Active development of other approaches, such as SLP and whole-function vectorization based on the VPlan architecture, has yet to be started.

References

[1] Diego Caballero and Intel Vectorizer Team. 2018. Extending LoopVectorize to Support Outer Loop Vectorization Using VPlan. https://llvm.org/devmtg/2018-04/slides/Caballero-Extending%20LoopVectorize%20to%20Support%20Outer%20Loop%20Vectorization%20Using%20VPlan.pdf Accessed: 2024-11-18.

[2] Florian Hahn, Satish Guggilla, and Diego Caballero. 2018. Outer Loop Vectorization in LLVM: Current Status and Future Plans. https://llvm.org/devmtg/2018-10/talk-abstracts.html#talk21 Accessed: 2024-11-22.

[3] Florian Hahn and Ayal Zaks. 2023. VPlan: Status Update and Future Roadmap. https://llvm.org/devmtg/2023-10/slides/techtalks/Hahn-VPlan-StatusUpdateAndRoadmap.pdf Accessed: 2024-11-19.

[4] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, USA, 75.

[5] Simon Moll and Sebastian Hack. 2017. VPlan + RV: A Proposal. https://www.llvm.org/devmtg/2017-10/slides/Moll-Vplan.pdf Accessed: 2024-11-19.

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

[7] Simon Moll, Thorsten Klößner, and Sebastian Hack. 2018. RFC: A new divergence analysis for LLVM. https://llvm.org/devmtg/2018-04/slides/Moll-A%20new%20divergence%20analysis%20for%20LLVM.pdf Accessed: 2024-11-19.

[8] Ashutosh Nema and Anupama Rasale. 2023. Improving Vectorization for Loops with Control Flow. https://llvm.org/devmtg/2023-05/slides/QuickTalks-May10/05-EuroLLVM23%20-%20Improving%20Vectorization%20for%20Loops%20with%20Control%20Flow.pdf Accessed: 2024-11-20.

[9] Rouzbeh Paktinatkeleshteri, João P. L de Carvalho, Ehsan Amiri, and J. Nelson Amaral. 2023. Efficient Auto-Vectorization for Control-flow Dependent Loops through Data Permutation. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering (CASCON '23)*. IBM Corp., USA, 74–83.

[10] Wyatt Praharenka, David Pankratz, João P. L. De Carvalho, Ehsan Amiri, and José Nelson Amaral. 2022. Vectorizing divergent control flow with active-lane consolidation on long-vector architectures. *Journal of Supercomputing* 78, 10 (July 2022), 12553–12588. https://doi.org/10.1007/s11227-022-04359-w

[11] Gil Rapaport and Ayal Zaks. 2017. Introducing VPlan to the Loop Vectorizer. https://llvm.org/devmtg/2017-03//assets/slides/introducing_vplan_to_the_loop_vectorizer.pdf Accessed: 2024-11-16.

[12] Hideki Saito and Intel Vectorizer Team. 2016. Extending LoopVectorizer towards supporting OpenMP4.5 SIMD and outer loop auto-vectorization. https://llvm.org/devmtg/2016-11/Slides/Saito-NextLevelLLVMLoopVectorizer.pdf Accessed: 2024-11-18.

[13] 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.

[14] LLVM Team. 2021. Auto-Vectorization in LLVM. https://llvm.org/docs/Vectorizers.html Accessed: 2024-11-18.

[15] LLVM Team. 2024. Vectorization Plan. https://llvm.org/docs/VectorizationPlan.html Accessed: 2024-11-18.

[16] Compiler Design Lab Saarland University. 2024. RV: A Unified Region Vectorizer for LLVM. https://github.com/cdl-saarland/rv Accessed: 2024-11-11.

[17] Ayal Zaks and Gil Rapaport. 2017. Vectorizing Loops with VPlan – Current State and Next Steps. https://llvm.org/devmtg/2017-10/slides/Zaks-Vectorizing%20Loops%20with%20VPlan.pdf Accessed: 2024-11-16.

A Appendix: Additional Figures

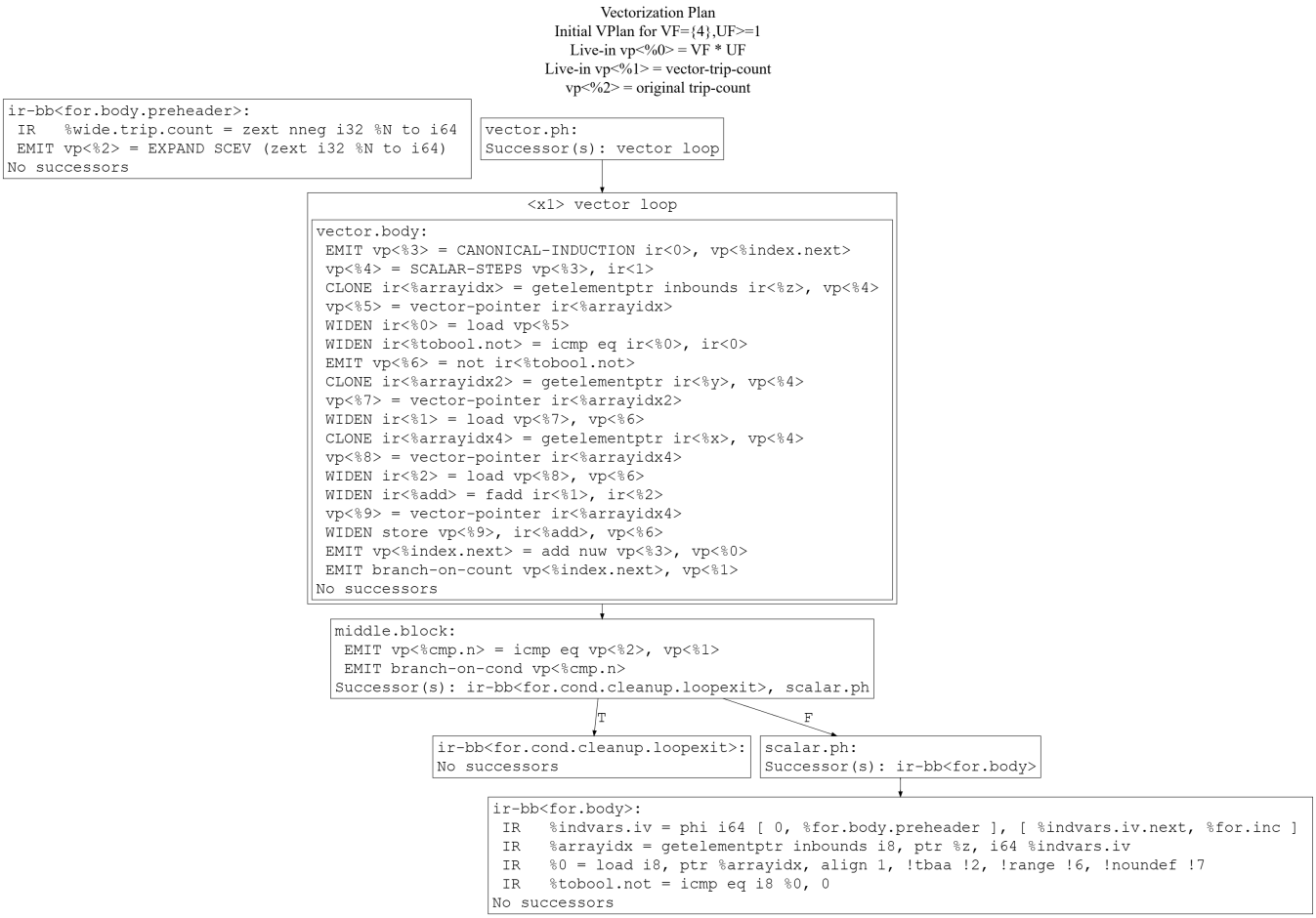

Figure 6. Full hierarchical CFG of a VPlan modelling the vectorized version of the loop depicted in Figure 1 as of LLVM 19.1.4.