Scalar Interpolation: A Better Balance between Vector and Scalar Execution for SuperScalar Architectures

Reza Ghanbari
University of Alberta
Edmonton, Canada
rghanbar@ualberta.ca

Henry Kao
Huawei Technologies Canada
Toronto, Canada
henry.kao1@huawei.com

João P. L. De Carvalho
University of Alberta
Edmonton, Canada
labegali@ualberta.ca

Ehsan Amiri
Huawei Technologies Canada
Toronto, Canada
ehsan.amiri@huawei.com

J. Nelson Amaral
University of Alberta
Edmonton, Canada
jamaral@ualberta.ca

Abstract

Most compilers convert all iterations of a vectorizable loop into vector operations to decrease processing time. This paper proposes Scalar Interpolation, a technique that inserts scalar operations into vectorized loops to increase the utilization of execution units in processors with distinct pipelines for scalar and vector processing. Scalar interpolation inserts scalar operations for an entire iteration of the sequential loop to avoid data movements between vector and scalar registers. A challenge to introducing scalar interpolation is creating a static cost model to guide the compiler's decision to interpolate scalar operations in a loop. An alternative to a static cost model is to perform auto-tuning in a loop to dynamically discover a sweet spot for the scalar interpolation factor. A performance study on an LLVM-based prototype reveals speedups of up to 30% on Intel Xeon (x86) with a static analysis of the cost model, and 43% on Kunpeng-920 (AArch64) with auto-tuning.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: Static Analysis, Loop Vectorization

ACM Reference Format:
Reza Ghanbari, Henry Kao, João P. L. De Carvalho, Ehsan Amiri, and J. Nelson Amaral. 2025. Scalar Interpolation: A Better Balance between Vector and Scalar Execution for SuperScalar Architectures. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3696443.3708950

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708950

1 Balancing Scalar and Vector Pipelines

Applications may exhibit data-level parallelism (DLP) and instruction-level parallelism (ILP). DLP, Single-Instruction Multiple-Data (SIMD) in Flynn's taxonomy, consists of executing a single operation on multiple data simultaneously, and ILP occurs when multiple independent instructions are executed concurrently. Vector instructions target DLP by processing many data elements with a single vector instruction. Conventional processors also exploit ILP and DLP by implementing multiple execution pipelines to handle multiple vector instructions per cycle.

Compilers can utilize these vector processing units by automatically generating vector code from scalar code. Commercial compilers use two general vectorization techniques: loop vectorization and Superword-Level Parallelization (SLP) [16]. Both use a cost model to determine whether or not applying vectorization yields performance improvement. The primary metric in such cost-model heuristics is instruction latency. Loop vectorization applies vectorization if the estimated latency of one iteration of a vectorized loop with vector width `N` is lower than the time required to execute `N` iteration of the scalar version of the loop. SLP applies vectorization to straight-line code if its cost model predicts that the additional estimated latency of packing/unpacking data to conform to vector code is less than the execution of the original scalar code. However, conventional compiler vectorization techniques fail to consider the utilization balance of hardware execution units. Most contemporary superscalar processors feature distinct pipelines for scalar and vector processing. The code generated by vectorization optimizations contains long dense sequences of vector instructions that put high pressure on the vector processing elements. Few scalar instructions are seen in the vectorized blocks of code, leaving the scalar units mostly idle while executing the

Concepts and techniques presented in this paper are subject matter of pending patent applications filed by Huawei Technologies.

vectorized code. Thus, there is an opportunity to use these idle scalar units to extract more ILP and DLP.

Partial vectorization is a proposed solution to tackle the scalar and vector resource-utilization imbalance by scheduling a subset of instructions to the scalar units within a loop iteration. However, partial vectorization requires extra data movement between scalar and vector registers where data use, such as a def-use dependence, crosses a boundary between scalar and vector instructions [5]. Extra data movements increase the pressure on memory and registers and reduce the cases in which partial vectorization is beneficial.

This paper proposes scalar interpolation, a new approach for loop vectorization that exploits the separate vector and scalar pipelines in out-of-order superscalar processors by selecting some of the iterations of a vectorized loop to be executed in the scalar pipeline. Scalar interpolation can more effectively use the execution units without imposing additional pressure on memory and registers. Scalar interpolation is always legal for vectorized loops because it does not change any data dependencies in the loop. However, a legality check is still needed because the vectorization pass may introduce loop-carried dependencies into the vectorized loop through reuse of vector registers. A scalar-interpolation code transformation requires a strategy to decide how many scalar iterations to interpolate (i.e., interleave) into a vectorizable loop. This paper presents two ways of determining the scalar-interpolation factor for a given loop: (i) an analytical cost model that is based on a simulation of a list-scheduling algorithm and the characteristics of the target processor architecture; and (ii) an auto-tuning approach that explores various scalar-interpolation factors to select the most beneficial one. The main contributions of this work are:

- Scalar Interpolation, a new vectorization technique that interpolates scalar iterations in a vectorized loop.
- A cost model for scalar interpolation based on a simulation of the scheduling of the loop execution and on processor specification.
- A prototype implementation of scalar interpolation and the cost model in LLVM.
- An experimental evaluation that determines that scalar interpolation leads to performance improvement in PolyBench [22].

2 Stalls Caused by Saturated Issue Queues

Conventional compiler vectorization aims to extract maximum ILP from vectorized code by saturating the available vector-processing resources. For instance, the current implementation of loop vectorization in LLVM vectorizes and interleaves loops: instructions from different loop iterations are interleaved to schedule the same instruction from different loop iterations in the same cycle. The number of interleaved loop iterations depends on the processor resources available for simultaneous use.

```
1  for (k = 0; k < _PB_M; k++)
2      C[i][j] += alpha * A[i][k] * A[j][k];
```

Listing 1. The loop in line 88 of `syrk` kernel in PolyBench benchmark after performing loop interchange

Benchmark `syrk` from the PolyBench suite [22] after performing a loop interchange will contain a vectorizable loop shown in Listing 1 that exemplifies vectorization with interleaving. First LLVM determines that the vectorization of this loop is legal and beneficial, and then it transforms the scalar loop to a vectorized version. Loop interleaving is applied to saturate the processor's vector units. Listing 2 shows the transformed pseudo-code after compilation for the TSV110 microarchitecture. The loop-vectorization cost model selects a vector width of four and an interleave factor of two. Table 1 shows the top-down analysis [31] of vectorized `syrk` performed on a Kunpeng 920 machine (TSV110 microarchitecture). Top-down is a performance analysis technique that identifies bottlenecks in CPUs by sampling various performance counters. Predefined formulas based on the available performance counters calculate what fraction of cycles are spent in different stages of the processor pipeline. Bottlenecks are identified by starting at the top-most and broadest level of categories (i.e., front-end and back-end bound) and narrowing down to more specific categories. `syrk` is mostly back-end bound. The CPU backend consists of the `Core` (i.e., execution units) and `Memory` (i.e., data accesses to memory) components.

```
1  vc1[0:3] = {0, 0, 0, 0};
2  vc2[0:3] = {0, 0, 0, 0};
3  for (k = 0; k < ⌊_PB_M/8⌋ * 8; k += 8) {
4      vc1[0:3] += alpha * A[i][k: k + 3]
5          * A[j][k: k + 3];
6      vc2[0:3] += alpha * A[i][k + 4: k + 7]
7          * A[j][k + 4: k + 7];
8  }
9  c[i][j] += sum_lanes(vc1 + vc2)
10 for (; k < _PB_M; k++)
11     C[i][j] += alpha * A[i][k] * A[j][k];
```

Listing 2. `syrk` kernel pseudo code after vectorization. The vectorized loop has an epilogue loop that is responsible for the remaining computations if the trip count is not a multiple of the vectorization factor.

The dominance of the `Core Bound` sub-category under `Backend Bound`, indicates that in `syrk` most of the stall cycles are due to execution units, and not cache misses. The high percentage of `Exec. Port Full` in `Core Bound` reveals that the processor stalls because the issue queues in the execution ports are full: instructions fill the queues faster than

Table 1. Top-Down Analysis of `syrk`.

Category	Value
Frontend Bound	19.7%
Bad Speculation	11.0%
Retiring	34.8%
Backend Bound	34.5%
Resource Bound	4.4%
Core Bound	62.0%
Divider Stall	0.0%
FSU Stall	0.3%
Exec. Port Full	61.7%
ALU/BRU Issue Q. Full	0.1%
LSU Issue Q. Full	2.6%
FSU Issue Q. Full	17%
Memory Bound	26.4%

they are drained. Thus, instructions that are ready to execute must wait for a slot to be freed in the issue queues. `Exec. Port Full` can further be dissected into three sources of issue-queue stalls, from: (1) the scalar integer and branch issue queues (`ALU/BRU Issue Q.`); (2) the floating-point and SIMD (i.e., vector) issue queues (`FSU Issue Q.`); (3) the load and store unit issue queues (`LSU Issue Q.`). These stalls are reported in the top-down analysis as a fraction of issue queue full stall cycles normalized to total CPU cycles. Full-vector issue queues contribute 17% of the stall cycles while the scalar integer and load-store execution units are mostly idle. This top-down analysis shows a utilization imbalance in the vectorized `syrk`: vector resources are over-utilized leading to issue-queue stalls while the scalar units are underutilized.

Scalar Interpolation addresses the utilization imbalance of CPU execution units imposed by conventional vectorization optimizations (e.g., loop vectorization). Scalar Interpolation interleaves scalar iterations of a loop to (i) reduce pressure on the vector processing units, and (ii) offload work to the underutilized scalar resources.

3 Scalar Interpolation

The main idea is to offload some of the computation to scalar units to reduce the pressure on vector unit by interpolating scalar iterations into vectorized loops. The number of interpolated scalar iterations is the scalar interpolation factor (*SIF*), determined by either a cost model or an auto-tuning process.

Listing 3 shows an example of a scalar-interpolation transformation applied to the loop in Listing 2. Adding scalar iterations to the vectorized loop reduces pressure on vector function units and increases ILP. Conventional vectorization causes stalls because it uses exclusively vector instructions thus saturating the vector units while leaving available scalar units unused. Scalar interpolation reduces this pressure in processors where the front-end dispatcher can forward the interpolated scalar operations to the scalar units while continuing to dispatch the vector operations without degrading the performance of the vector units.

```
1  vc1[0:3] = {0, 0, 0, 0};
2  vc2[0:3] = {0, 0, 0, 0};
3  vcs = 0; (4)
4  for (k = 0; k < ⌊_PB_M / 9⌋ * 9 (2); k += 9 (2)) {
5      vc1[0:3] += alpha * A[i][k: k + 3]
6          * A[j][k: k + 3];
7      vc2[0:3] += alpha * A[i][k + 4: k + 7]
8          * A[j][k + 4: k + 7];
9      vcs += alpha * A[i][k + 8] * A[j][k + 8]; (1)
10 }
11 c[i][j] += sum_lanes(vc1 + vc2) + vcs (3) (4);
12 for (; k < _PB_M; k++)
13     C[i][j] += alpha * A[i][k] * A[j][k];
```

Listing 3. syrk benchmark pseudo code when scalar interpolation is applied with SIF=1. Highlighted code are modifications that scalar interpolation make

3.1 Legality Check

Vectorization is applied to loops free of loop-carried dependencies or when the dependence distance is larger than the vectorization factor. However, the vectorized loop may have loop-carried dependencies introduced by the vectorizer. For instance, when generating code for a loop with a first-order recurrence, LLVM reuses the vector registers containing the loaded values of the recurrence from the previous iterations to avoid memory operations, leading to loop-carried dependencies in the vectorized loop.

Thus, the scalar-interpolation legality check ensures no dependency between the instructions scheduled to the scalar unit and vector instructions. This constraint: (i) avoids the significant overhead associated with data transfers between vector and scalar registers and related memory interactions; and (ii) exposes ILP opportunities for subsequent scheduling passes, such as software pipelining.

The legality-check pass avoids loops with control flow because the scalar interpolation algorithm does not support it. Some compilers support vectorization of control flow in loops by flattening the control-flow graph using predicated instructions for processors that support predicates. The algorithm can be extended to handle if-converted control flow for such processors. The modified algorithm would determine when interpolated masked instructions are needed.

Finally, the legality check excludes loops having floating-point operations. Scalar interpolation is limited to integer instructions for now because, in many architectures, vector

and floating-point units share the same ports or pipelines. Consequently, replacing vector floating-point instructions with scalar ones would not improve performance. The algorithm is independent of the instruction types and can be extended to support floating-point types.

3.2 Scalar-Interpolation Transformation

This description of a scalar-interpolation transformation assumes that it is a pass in a compiler intermediate representation that uses Static-Single Assignment (SSA) where variables have been renamed to ensure the SSA property and distinct values are merged by ϕ instructions at join points.

As demonstrated in Listing 3, to preserve program semantics, the scalar-interpolation transformation must: (1) interpolate scalar instructions using the SIF derived from the cost model; (2) adjust the step and termination condition of the loop induction variable to reflect the modified loop iterations; (3) handle live-out values correctly; and (4) manage reduction operations.

Algorithm 1 shows the scalar-interpolation transformation algorithm with the vectorized loop body and the SIF as inputs. The algorithm goes through each vector instruction in the loop and inserts *SIF* equivalent scalar instruction as shown in lines 19 to 26.

To preserve loop-independent dependencies, Algorithm 1 updates *InstrMap* — which maps a vector instruction to its corresponding interpolated instructions — whenever an instruction is interpolated. Interpolated-instruction operands are selected from the interpolated instructions stored in this mapping. Algorithm 1 processes the loop instructions in reverse post order to ensure that it visits all the dependencies of an instruction before visiting the instruction itself. This way, *InstrMap* always contains all the dependencies of an instruction being interpolated.

The Scalar Interpolation Factor (SIF) is the number of iterations of the original loop that are interpolated. To adjust the step and termination condition, as illustrated in Listing 3, the induction-variable update operation is extracted from the original instruction, and *SIF* instructions are inserted, each updating the induction variable by 1. SIF update instructions are required because, to preserve dependencies, each interpolated iteration accesses its own update instruction. This is done in line 18 of Algorithm 1. After vectorization and scalar interpolation with factor *SIF*, the loop step is $S \times (UF \times VF + SIF)$, where S is the original step and UF and VF are unrolling and vectorization factors respectively.

The *handleLiveOuts* function collects all def-use relations where the definition occurs inside the loop and the use is outside. For def-use pairs, it updates the definition to refer to the last equivalent interpolated instructions for the definition using the *InstrMap*. This last instruction produces the final live-out value for the definition. If the loop trip count is not known at compile time or if it is not a multiple of $UF.VF+SIF$, the live-out values are computed by the epilogue loop and are not a concern for scalar interpolation.

A reduction operation contains a ϕ instruction that merges an initialization value and a reinitialization value from the loop body. The vectorized loop contains such a ϕ instruction that received the updated value from a vector instruction. Interpolated scalar iterations have a similar ϕ instruction with the same initialization (lines 8 to 17 of Algorithm 1) but the source of the values are scalar instructions. Thus, for correct code generation, the scalar-interpolation instructions for the loop body must be generated before the edges for this ϕ instruction are completed by the *FixPhis* function.

At the end of the transformation, the algorithm aggregates the scalar reductions with the reduced value of vector ones in the loop exit block to generate the final reduction value.

Algorithm 1 Scalar Interpolation Transformation

```
1:  procedure Transform(SIF, Loop)
2:      IP ← NULL  ▷ insertion point for new instructions
3:      InstrMap ← {}
4:      PhisToFix ← []
5:      IVs ← Loop.getInductionVariables()
6:      for BB in reversePostOrder(Loop) do
7:          for Instr in BB do
8:              if Instr instanceof ReductionPhi then
9:                  PhisToFix.append(Instr)
10:                 IP ← Instr
11:                 for i = 0 to SIF − 1 do
12:                     ScalarInstr ← cloneScalar(Instr)
13:                     ScalarInstr.insertAfter(IP)
14:                     IP ← ScalarInstr
15:                     InstrMap[Instr].append(
                                ScalarInstr)
16:                 end for
17:             end if
18:             IP ← insertUpdateInstrsForIV(
                        Instr, InstrMap, SIF, IP, IVs)
19:             for i = 0 to SIF − 1 do
20:                 Operands ← InstrMap.get(
                                Instr.operands(), i)
21:                 ScalarInstr ← cloneScalar(Instr)
22:                 ScalarInstr.setOperands(Operands)
23:                 ScalarInstr.insertAfter(IP)
24:                 IP ← ScalarInstr
25:                 InstrMap[Instr].append(ScalarInstr)
26:             end for
27:         end for
28:     end for
29:     FixPhis(PhisToFix, Loop, InstrMap, SIF)
30:     handleLiveOuts(Loop, InstrMap, SIF)
31: end procedure
```

3.3 Cost Model

The cost model seeks to predict a Scalar Interpolation Factor (SIF) that balances the workload distribution between scalar and vector units and increases overall execution unit utilization. The goal is to interpolate as many scalar iterations as possible without increasing the execution latency of the original vector loop. The cost model uses a critical-path-first greedy list scheduling algorithm [14] for instruction sequencing to estimate the schedule length of the loop as a proxy for execution latency. This scheduling algorithm is widely adopted as the main instruction scheduler by production compilers [1, 2].

Algorithm 2 first estimates L_{vec}, the schedule length of the original vectorized loop using the Data Dependence Graph of the loop body. Then, it keeps increasing the SIF, starting from 1, to select the highest SIF that does not increase the schedule length beyond L_{vec}.

A processor execution engine consists of different execution ports, each supporting a subset of instruction types. The SIF estimation must take port utilization into account. The solution is to assign a versatility score V_i for each port i based on the support of its instruction types by other ports. Let N be the number of ports in the architecture, N_j be the number of ports that support instructions of type j, and K_i be the number of instruction types supported by port i. Equation 1 calculates V_i by summing the number of ports that do not support instruction types that are supported by port i.

$$V_i = \sum_{j}^{K_i} (N - N_j) \tag{1}$$

When there is a choice of ports to assign to a given instruction, the strategy for the list-scheduling algorithm in the cost model is to aim to keep more versatile ports free for later scheduling. Thus, a port with lower V_i has a higher priority for scheduling.

Applying the cost model algorithm to the loop in Listing 4 with the port mapping in Table 2 yields a initial vector-loop schedule with 14 cycles as shown in the darker operations in Figure 1a.

Figure 1a shows a schedule with SIF=1 that follows Equation 1 for port assignment. This schedule increases the execution latency of the loop from 14 to 17 cycles leading the cost model to reject any SIF ≥ 1. However, SIF=1 can be achieved within the 14 cycles of the original vector schedule by changing port assignments as shown in Figure 1b.

To discover better schedules with alternative port assignments, the *applyListScheduling* procedure attempts alternative schedules that use a randomized scheduling priority R_i for each port given by Equation 2 where $0 \leq u \leq 1$ is a random variable with uniform distribution, and c is a counter starting from 1 and increasing for each trial until a fixed limit (80 in the prototype). The procedure returns the best schedule found.

Algorithm 2 Scalar Interpolation Cost Model

```
1:  procedure SelectBestSIF(Loop, Spec, CM, Budget,
    StablilityLimit)
2:      SIF ← 0
3:      BestScheduleLen ← ∞
4:      while True do
5:          DAG ←constructDAG(Loop, SIF)
6:          GreedySched ← applyListScheduling(
                DAG, Spec, Budget, StablilityLimit)
7:          if BestSchedLen < GreedySched.length then
8:              return SIF − 1
9:          end if
10:         if SIF = 0 then
11:             BestSchedLen ← GreedySched.length
12:         end if
13:         SIF ← SIF + 1
14:     end while
15: end procedure
```

Table 2. The execution engine details of the architecture on which the example code is running.

Port	0	1	2	3	4	5
Units	Int ALU Vec ALU Vec MUL Int DIV Branch	Int ALU Vec ALU Vec MUL Int MUL	LD ST	LD	Int ALU	Int ALU Vec ALU
V_i	19	14	9	4	2	5

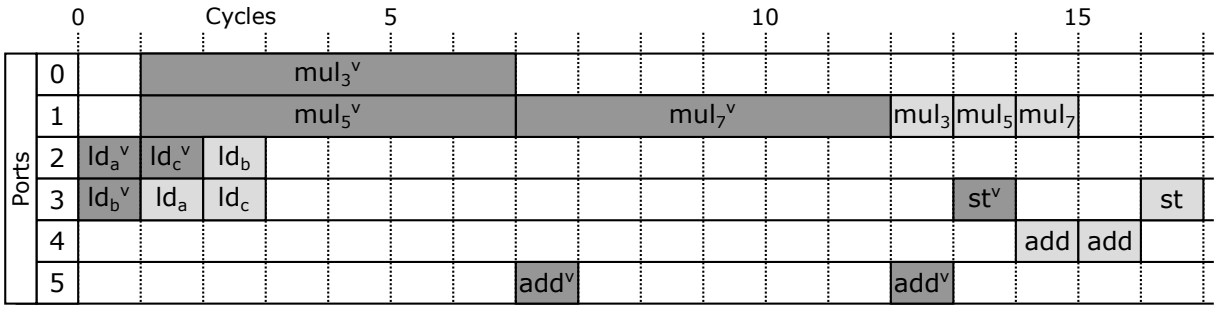
$$R_i = \frac{c-1}{cV_i} + \frac{u}{c} \tag{2}$$

The rationale for Equation 2 is to create a strategy similar to simulated annealing [15]. For early trials, there is significant randomization in the priority of each port and, as the algorithm advances, the priorities stabilize around the value of $\frac{1}{V_i}$. The scheduling algorithm selects the candidate port with higher R_i. To prevent the algorithm from spending too much time attempting to improve a good schedule, a stabilization threshold (set to 10 in the prototype) is used. If a better schedule is not found in the number of consecutive trials within this threshold, the algorithm returns the best schedule found.

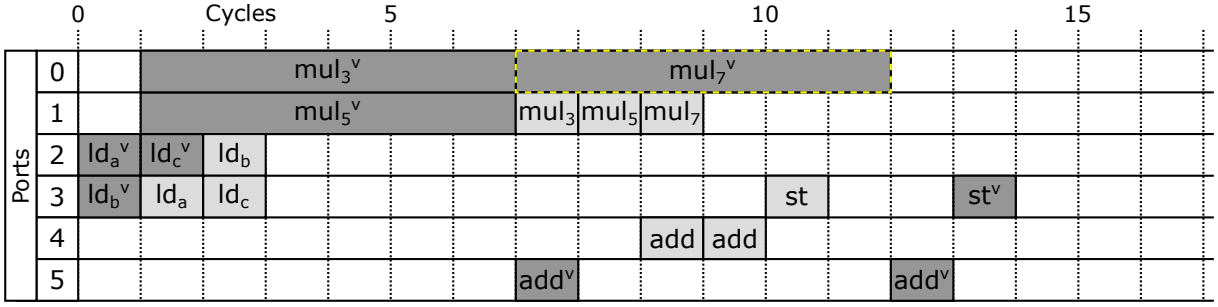
For Listing 4, the improved schedule, found using the Equation 2 strategy, is shown in Figure 1b. It maintains the same execution latency of the original vector loop, while utilizing scalar units to reduce the total execution time of the loop.

```
1  for (int i = 0; i < N; i++)
2      f[i] = (3 * a[i]) + (5 * b[i]) + (7 * c[i]);
```

Listing 4. A vectorizable loop amenable for Scalar Interpolation.



(a) A naive schedule generated following the versatility score



(b) One of the suggested schedules of the cost model

Figure 1. Schedules for Listing 4 with SIF of 1. Vector instruction are denoted with a "v" superscript in dark-grey boxes. Subscripts denote which source code the instruction is derived from (e.g., ld_a^v – a vector load from array a). Scalar instructions are in light-grey boxes (e.g., mul_3 – a scalar multiply by 3).

3.4 Auto-Tuner

Currently scalar interpolation is only applied to loops without control flow. Thus, even though the iteration count for each loop may depend on the program's input, the best scalar interpolation factor (SIF) is independent of a program's input and can be determined empirically using an auto-tuning strategy.

The auto-tuner takes a set of pre-defined SIF values as input. The program is compiled with each of the given SIF values and also without scalar interpolation. Each variant is executed and the loop-execution time is measured – loops that run for a very short amount of time can be placed in an external loop to allow more precise time measurement. To account for variations in execution time because of external variables, the measurement is repeated a fix number of times. The auto-tuner returns the value of SIF for the most performant version. For the experimental evaluation in Section 5.4, each experiment is repeated ten times and the set of values of SIF tested by the auto-tuner is 1, 2, 4, 8.

4 A Prototype Integration of Scalar Interpolation in a Vectorization Plan

We implement scalar interpolation in LLVM 17 within the loop-vectorization pass — a target-agnostic pass in the mid-end of the optimization pipeline that operates on LLVM Intermediate Representation (IR). The decision to implement Scalar Interpolation as a feature of loop vectorization is based on several design points: (i) Scalar Interpolation is only useful when applied on the vectorized code, thus the cost model should have information about the vector loop readily available. (ii) There is a trivial mapping between each IR instruction (or a group of IR instructions) in the original loop and their vectorized counterparts in the planning step (available through Vectorization Plan Recipes [4]). (iii) Scalar Interpolation should be a target-agnostic transformation, thus it should be implemented in a target agnostic mid-end pass.

A mid-end implementation lacks precise target-machine information (e.g., instruction latency and available execution ports). Scalar Interpolation could also be implemented after instruction selection where the LLVM IR is lowered to Machine IR (MIR). The more precise target information available at the MIR level would improve the scalar-interpolation cost-model predictions. However, the engineering effort would be significant because scalar instructions and vector instructions no longer have a trivial mapping to one another. Thus, the mid-end prototype of Scalar Interpolation uses an estimated latency for each LLVM IR instruction from LLVM's Target Transform Info (TTI) [3]. The cost model combines these latencies with resource availability from processor-architecture documentation to generate the expected schedule needed to predict SIF.

5 Experimental Evaluation

This experimental evaluation using the LLVM-based prototype aims to address the following research questions: ① Is the proposed cost model effective in predicting an interpolation factor? ② Does scalar interpolation result in performance improvements? ③ Does scalar interpolation increase functional-unit utilization?

5.1 Computers, Compilers, and Benchmarks

The experimental evaluation used two machines: (i) an x86_64 machine running Ubuntu18.04.3 LTS (kernel version 4.15.0) on Intel Xeon E5-2698 v4 locked at 2.2GHz; and (ii) an AArch64 machine running OpenEuler 22.03 (kernel version 4.19.90) on Kunpeng-920 locked at 2.6GHz.

This evaluation uses the PolyBench/C benchmark suite [22] that contains numerical computations such as linear algebra kernels and operations that expose many vectorization opportunities. The preprocessor macros of PolyBench are used to set the data type to int and set the dataset size to extra large. Some benchmarks are adapted for integer computation while preserving semantics. For instance, a multiplication by 0.125 is replaced by a shift right in `heat-3d`. The array initialization process is modified to adapt to integer computation. Originally, array elements were set using a modulus operation with the array size followed by a division by the array size, resulting in zero values when compiled for integers. To prevent the execution-time distortion created by zero propagation, the division was removed to ensure non-zero initial values for the arrays.

The experimental evaluation measures the elapsed CPU time using the `clock()` function. Time measurement for loops that run for very short run times is imprecise. This study does not report results for individual loops that run for less than 10 milliseconds and benchmarks that run for less than 100 milliseconds. Each benchmark is compiled with Clang version 17.0.0 with the flags `-O3 -fno-slp-vectorize`. The second flag disables SLP vectorization to prevent the SLP vectorizer from undoing the changes made by scalar interpolation.

5.2 Experimental Methodology

Two configurations are tested in all the experiments in this section: (i) **Baseline:** The benchmark is compiled with Clang version 17.0.0 using LLD as the linker, and the flags mentioned in the Section 5.1. (ii) **Scalar Interpolation:** For the prototype implementation of the scalar interpolation transformation, detailed in Section 3, we created two new clang pragmas, one to enable/disable interpolation and the other to specify the value of SIF.

Section 5.4 contains two types of experiments: (i) innermost-loop evaluation; and (ii) whole-benchmark evaluation applies scalar interpolation to all eligible loops. In both cases, an average of ten repetitions of the measurement is reported. Section 5.5 reports a study, using the perf tool, of the best-performing benchmark on each target. This study reports the average number of cycles from ten executions of each of these benchmarks. In Section 5.3, the cost model prediction is compared to the auto-tuning approach.

5.3 The Cost Model Effectiveness Is Limited

For question ①, the evaluation contains investigating 93 scalar-interpolation candidate loops in the Intel Xeon target. In 25 loops the cost model predicts the best performing SIF. In 53 loops, either the execution time is too short or the performance of the cost-model-based transformation is within 1% of the performance of the auto-tuned version. The auto-tuner outperforms the cost model for the remaining fifteen loops shown in Figure 2b.

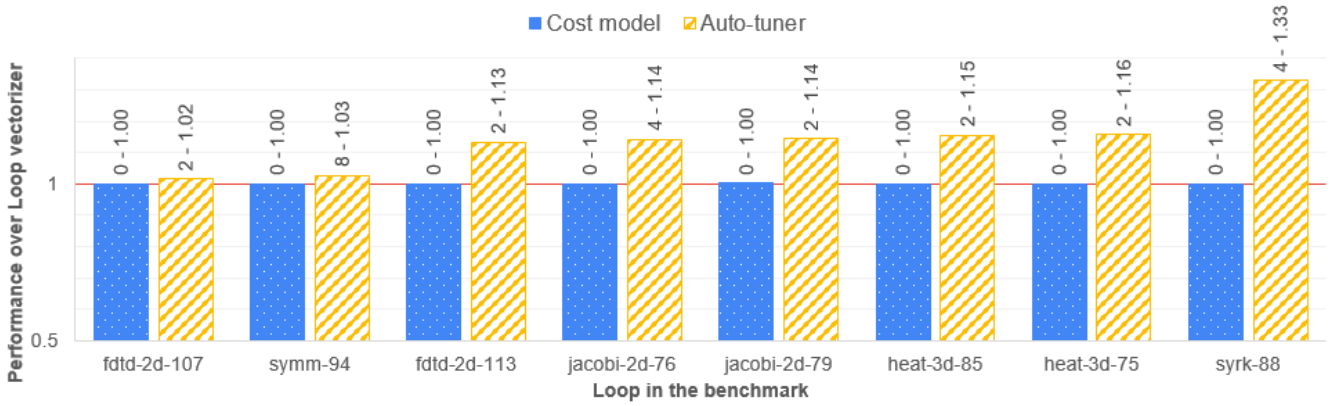
The cost model overestimates the SIF in eleven loops. In `choleski-49`, `lu-50`, `3mm-105`, `trmm-87`, `2mm-100`, `ludcmp`, and `doitgen-77` a dot-product computation is performed with column-major access to at least one of the involved tensors.[1] The compiler uses scalar loads to fetch the values of the column-major tensor and then uses vector-packing instructions to create a vector from the loaded elements. Applying scalar interpolation with high SIF in these cases puts even more pressure on the load/store units of the target.

In addtion to that, in `gemm-93` a vector is multiplied with the constant value 3. The later code generation lowers this multiplication into two additions. However, the cost model, computed from the LLVM IR representation, accounts for multiplications thus overestimating the cost of each iteration. In an experiment designed to confirm this hypothesis, When the constant is changed to 13 to prevent this back-end optimization, the cost model predicts the best-performing SIF. Future implementations may improve the cost model by modeling later optimizations.

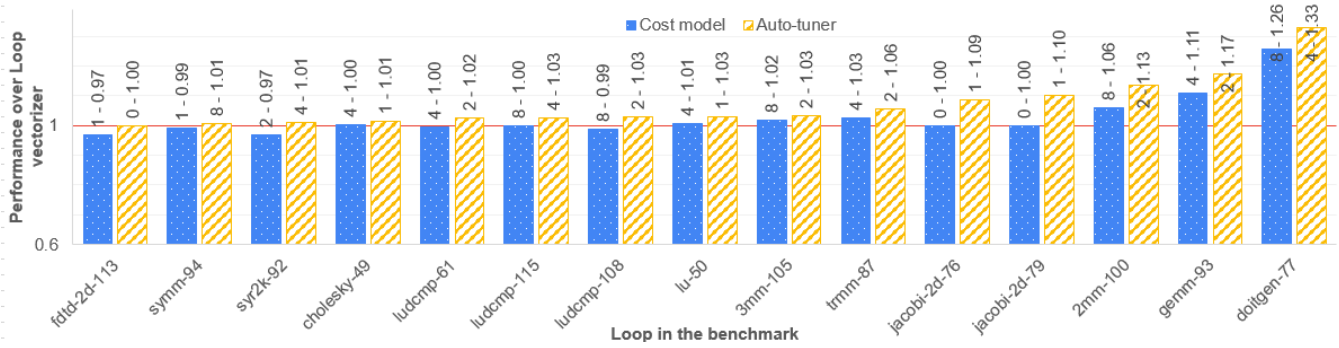
The cost model conservatively avoids scalar interpolation if the schedule length increases. However, for some loops such increases may be beneficial. For instance, in `jacobi-2d-76` and `jacobi-2d-79` the initial vector schedule length is 12 cycles, and a single-iteration interpolation one extends it to 13 cycles but results in better performance with a vectorization factor of four.

In the prototype, the cost model relies on the LLVM target-specific cost estimate for each statement. For the Kunpeng target, the higher latency of vector load/store instructions, relative to scalar ones, is not reflected in the LLVM cost estimates. Thus, the cost model predicts a SIF of zero for all loops that run for more than ten milliseconds. However, scalar interpolation increases performance in 15 loops for this target as discovered by the auto-tuner and shown in Figure 2a. For

[1] A loop is designated by the benchmark name followed by the loop line number in the code.



(a) Kunpeng target. The performance axis starting point is 0.5.



(b) Intel Xeon target. The performance axis starting point is 0.6.

Figure 2. Cost model performance compared to the auto-tuner. Each bar is annotated with the suggested SIF of the approach plus the performance over the baseline in form of SIF-performance.

`Jacobi-2d-76`, `Jacobi-2d-79`, and `fdtd-2d-113`, the auto-tuner discovers that an increase of two cycles to the length of the schedule — avoided by the cost model — is beneficial.

5.4 Scalar Interpolation Improves Performance

Two experiments that compare the performance of scalar interpolation to the baseline address question ②: (i) loop-level execution time; and (ii) whole-benchmark execution time. Given the limited effectiveness of the cost model for the Kunpeng target, this section reports the results from the cost model for the Xeon target and the results from the auto-tuner for the Kunpeng target.

On the Intel Xeon target, the transformation is applied on 92 loops of PolyBench. For 72 loops either the execution time is very short or the performance effects of scalar interpolation are negligible. The remaining 20 loops are represented on Figure 3. On the Kunpeng target, 43 candidates are found for scalar interpolation, and among them, the auto-tuner discovers a SIF that results in performance gains for the eight loops shown in Figure 4.

The number of loops in which scalar interpolation is applied are different in two targets and the loops in the charts are not the same in most cases, because loop vectorizer and scalar interpolation heavily depend on target-specific information for their cost model. Therefore, some vectorizable loops that are deemed beneficial in one target might not be considered beneficial in another target. Same can happen for scalar interpolation given that the cost model uses the LLVM target-specific cost model to access the estimation of latency of instructions. Moreover, the charts only show the results for the loop with a performance difference, which could be a different set of loops in different targets depending on the computation and the target capabilities. Another important point is that more than one scalar iteration can be executed while a vector iteration executes. Therefore, for the experiments shown in Figures 2, 3, and 4, the theoretical maximum speedup is $\frac{SIF}{VF} \times 100\%$.

An in-depth analysis of the results of this experimental evaluation indicates that there are two main causes for performance improvement due to scalar interpolation: loops with low memory interaction, and loops with costly vector operations.

Low memory interaction. In both targets, scalar interpolation is not beneficial for memory-bound loops because vector and scalar memory instructions use the same execution port. Thus, interpolating scalar iterations does not

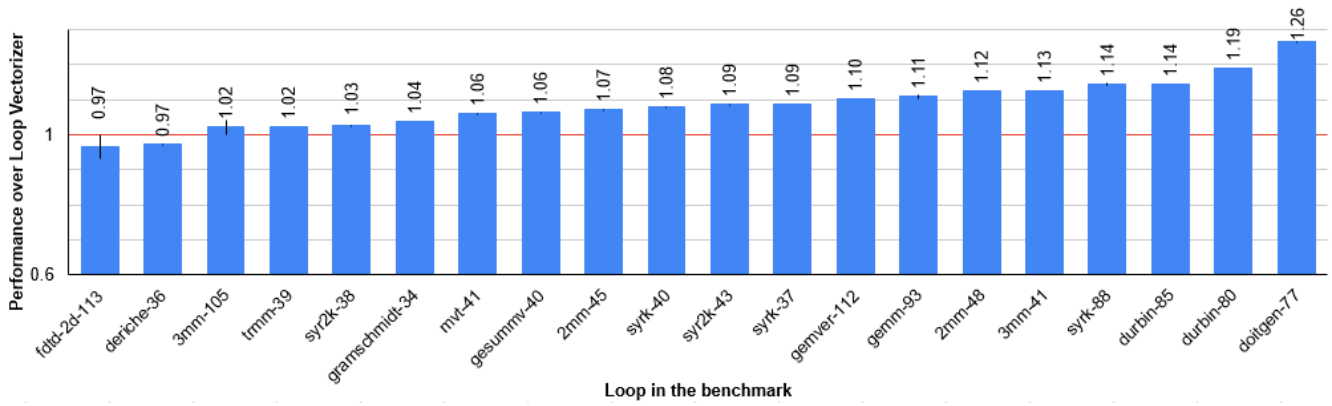


Figure 3. Performance results on the Intel Xeon. The performance axis starting point is 0.6 and the error bars show 95% confidence interval

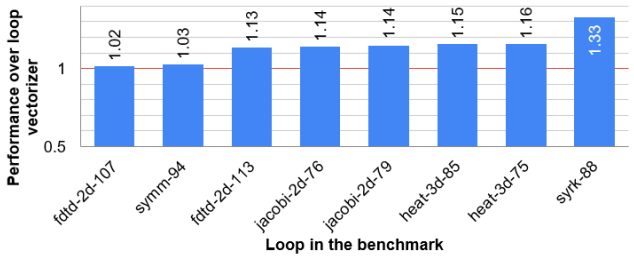


Figure 4. Performance results on Kunpeng. The performance axis starting point is 0.5.

affect the memory-unit utilization. The integer and the vector ALU units use separate ports on the architecture leading to benefits from interpolation for arithmetic-intensive loops. For instance, `syrk-88`, which gains the most benefit from scalar interpolation, is a reduction operation that computes a dot product of two rows of a matrix scaled by a constant factor. With an interleave count of two, the resulting loop contains four loads, four multiplications, two additions, and no store instructions leading to better utilization after scalar interpolation. The same pattern is observed on the loops in `heat-3d`.

Costly vector operations. Loop body contains vector operations that take longer to execute than their scalar counterpart, such as division. For instance, on Intel Xeon, the LLVM target-specific cost of a vector division by a constant is six times the scalar counterpart. In such cases scalar interpolation leverages scalar units that would have been idle. Of the loops in Figure 3, `fdtd-2d-113`, `jacobi-2d-76`, `jacobi-2d-79`, `doitgen-77`, `gemm-93`, and `3mm-105` exhibit this behavior. The last four loops compute matrix multiplication where one of the matrices is accessed in column-major order leading the loop vectorizer to load the matrix elements individually and then use a vector-packing instruction, which adds a noticeable overhead to the vector units.

Some initialization loops combine costly operations with low memory operations where scalar interpolation is beneficial, but they do not change the overall benchmark performance because initialization loops are a cold code region. These loops are composed of adds, multiplies, and costly modulo operations. The only memory interaction is the final store to the array element. Examples include `3mm-41`, `2mm-48`, `syrk-37`, `syr2k-43`, `syrk-40`, `2mm-45`, `gsummv-40`, `mvt-41`, `syr2k-38`, `gramschmidt-34`, and `trmm-39` on Figure 3. This effect is more pronounced in the Intel Xeon architecture where the LLVM cost model indicates that the cost of a vector multiplication with VF=4 is 6× that of a the scalar counterpart.

In `derich-36` and `fdtd-2d-113` on Figure 3 scalar interpolation results in a slow down. In `derich-36` the later code generator replaces a modulo operation by 65536 with a logical and, thus removing the costly modulo operation that led the scalar-interpolation cost model to predict a SIF = 8. This high SIF led to an increase in backend store-buffer stalls from 201 M/s (millions per second) to 204 M/s. In `fdtd-2d-113`, the cost model predicts SIF=1. Interpolating a single iteration leads to misaligned memory operation overheads. In this case the number of misaligned micro-operations dispatched to the L1 cache increases from 40.76 M/s to 66.79 M/s for loads and from 13.71 M/s to 21.36 M/s for stores. A future implementation should take memory-alignment into account in the cost-model SIF prediction. It can be achieved by adding another step to the cost model to apply alignment analysis and select the largest aligned SIF that is less than the SIF found by the current cost model.

For the whole-benchmark evaluation, the baseline SLP vectorizer is enabled to compare scalar interpolation with the entire LLVM vectorization pipeline. Only benchmarks with performance differences are shown in Figure 5 with up to 43% improvements in the performance of benchmarks. The whole-benchmark slow down in `fdtd-2d` for Xeon is because scalar interpolation causes memory misalignment

in the only loop for which the cost model result in a non-zero SIF (SIF-1 for the hot `fdtd-2d-113` loop).

5.5 A More Balanced Utilization

Question ③ requires an in-depth analysis of benchmarks where scalar interpolation results in speed improvement. This section presents results obtained from performance-counter measurements for `doitgen` on Xeon and `heat-3d` on Kunpeng.

The `doitgen` benchmark has three candidates but the cost model only predicts a non-zero SIF for two of them: SIF=4 for the initialization `doitgen-37` and SIF=8 for the hot `doitgen-77` that contains a reduction operation. Port utilization, reported in Figure 6 and measured using PMU counters, measures the rate at which micro-operations are issued for each port in the processor. Port 5 experiences a significant utilization decrease because it hosts vector packing and vector ALU units. Ports 2 and 3 are more utilized because they contain the execution units for the load micro-operations. Ports 0, 1, 4, and 5 are integer ALU ports. Scalar interpolation generates a more balanced micro-instruction distribution in this processor.

In `heat-3d` scalar interpolation is applied to the initialization loop `heat-3d-34` and the hot loops `heat-3d-75` and `heat-3d-85` in the Kunpeng target. In Kunpeng the micro-operations of scalar and vector instructions go through separate issue queues. Figure 7 shows that scalar interpolation increases the rate of load/store (LSU_IssueQ) and scalar (ALU/BRU_IssueQ) issue queue stalls because it increases scalar arithmetic/memory instructions. With VF=4 and SIF=2 in the transformed hot loops, in each iteration, four array elements are processed by vector instructions, and two by scalar instructions. Therefore, scalar interpolation decreases the vector-issue queue stalls (FSU_IssueQ) by 74%.

6 Related Work

Partial vectorization. Larsen *et al.* [5] introduce an approach for partial vectorization of the loop by splitting the computation over scalar and vector resources of a target hardware using a min-cut partitioning approach. Their cost function accounts for resource usage of each partition by calculating ResMII and the cost of transferring results between vector units and scalar units as needed. The main difference between their work and scalar interpolation is that they partition the loop body operations into scalar and vector sets, which causes the overhead of transferring data between scalar and vector registers and also requires additional memory operations in architectures without support for transferring data between scalar and vector registers. Scalar interpolation does not create dependencies between scalar and vector instructions, because it interpolates an entire iteration of a loop.

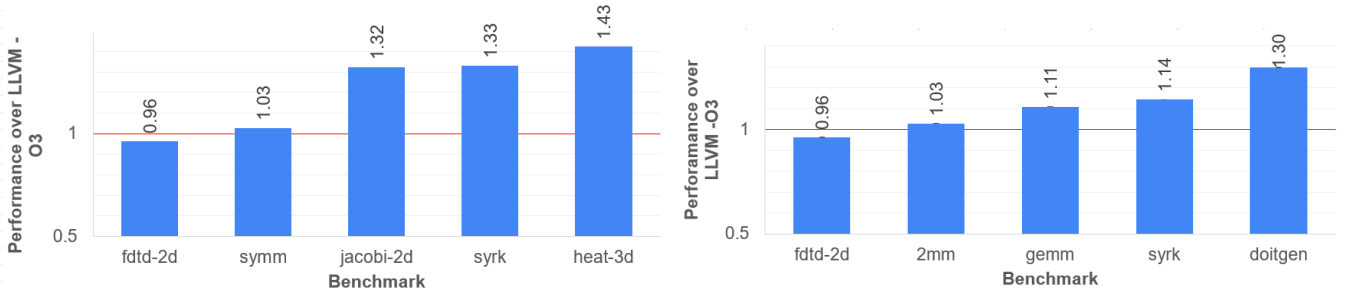
Rocha *et al.* [23] proposed an approach to select the unrolling factor of loops based on Straight-Line Parallelization (SLP) opportunities. They create a "Potential SLP Graph" based on the def-use relation of scalar instructions of the rolled loop, and then select the unrolling factor based on the Potential SLP Graph to maximize the utilization of vector units. Their approach is similar to loop unrolling to expose ILP opportunities in scalar interpolation. However, their goal is to find an unrolling factor that creates more vectorization opportunities for the SLP vectorizer. In contrast, the scalar-interpolation cost model is designed to better exploit both scalar and vector units of the processor by adding the scalar instructions to the vectorized loop.

SLP vectorization. Introduced by Larsen *et al.* [16], SLP vectorization targets basic blocks instead of loops and packs memory instructions with adjacent references into groups. This vectorization technique traverses the def-use chains in the program using the instruction groups as seeds to create a tree of groups of independent isomorphic instructions. All the groups in the tree are vectorized only if the benefit of vectorization outweighs the cost of instruction packing/unpacking. Several improvements [18–21] have been proposed to this algorithm, including the work of Shin *et al.* [26] that extends SLP to handle control flow.

Rosen *et al.* [24] discussed the idea of loop-aware SLP which is a hybrid approach between loop and SLP vectorization. Their approach is to decide whether to consider each instruction of the loop as a part of the SLP computation tree or to loop-vectorize them, and they discussed several factors that impacted this decision, such as alignment considerations and unrolling factor. Chen *et al.* [10] suggest SuperVectorization, an SLP-based vectorization that uses predicated SSA form to flatten the code and apply the SLP vectorization. By removing control-flow constructs they can combine loop and straight-line vectorization.

SLP can generate partially vectorized code based on a cost model where the benefit of a transformation is the savings of vectorized instructions minus the cost of packing and unpacking instructions regardless of the availability of vector resources. In contrast, the scalar-interpolation cost model embeds resource availability into the cost model. The SLP can be applied to any basic block in the program while scalar interpolation focuses only on the loops.

Unrolling. Unrolling loops by a given factor is a well-known code transformation [8, 9, 11, 25, 29]. Determining the unroll factor is important for performance and has been the focus of several studies [17, 27, 28]. In most compilers, the cost model used to select the unrolling factor computes a trade-off between the ILP introduced by unrolling versus code growth. In scalar interpolation, the loop is unrolled by the Scalar-Interpolation Factor (SIF), the instructions are reordered to expose both scalar and vector instructions to the target hardware, and the cost model is based on a scheduling algorithm that considers available hardware resources.



(a) Performance results on Kunpeng target

(b) Performance results on Intel Xeon target

Figure 5. Performance results for the whole benchmark. The performance axis starts from 0.5

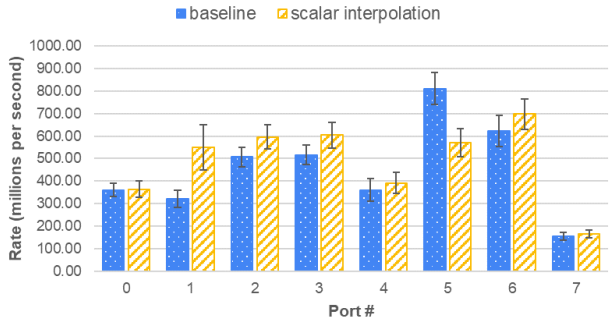


Figure 6. Port utilization on `doitgen` benchmark in Xeon. The error bars show 95% confidence interval

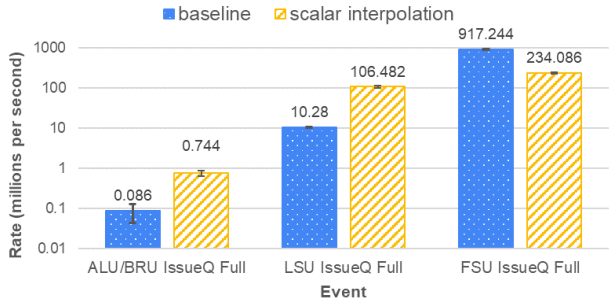


Figure 7. Port utilization on `heat-3d` in Kunpeng. The error bars show 95% confidence interval, and the chart is log-scale due to the significant difference of stalls between vector and scalar units.

Instruction scheduling. Instruction scheduling is a well-established and extensively studied problem. The general problem of instruction scheduling is known to be NP-Complete [29], and several heuristics have been proposed to provide near-optimal solutions to the problem [6, 7, 12, 14, 30]. Amongst them, different variations of list scheduling [14] have been widely adopted in production compilers [1, 2]. Scalar interpolation uses list scheduling in our implementation of the cost model to get an estimation of the schedule length.

7 Conclusion

Automatic vectorization passes in modern compilers are almost always enabled by default to exploit inherent ILP and DLP present within code. One such pass is loop vectorization – a transformation of scalar loops into vector equivalents. Although loop vectorization provides considerable performance benefits, we observed that conventional cost models failed to consider the balance in utilization between vector and scalar processing resources in superscalar out-of-order CPUs. The loop-vectorization cost model simply aimed to saturate vector processing units on the CPU to maximize ILP of vector instructions — potentially creating a throughput bottleneck on the vector resources. Our insight is that more throughput can be gained by offloading work from over-utilized vector resources onto the under-utilized scalar resources. Thus we propose a code transformation technique called Scalar Interpolation to address the imbalance of utilization between the scalar and the vector resources by interpolating scalar iterations into a vectorized loop. We implement a prototype for Scalar Interpolation in LLVM's loop vectorization pass and evaluate using PolyBench on two different CPUs with distinct ISAs. Scalar Interpolation effectively balances utilization of scalar and vector resources providing additional work throughput, resulting in speedups of up to 43% on Kunpeng-920 (AArch64) and 30% on Intel Xeon (x86) architecture.

A Artifact Appendix

A.1 Abstract

This artifact is a Docker image that contains all the requirements and scripts to replicate the experiments of the sections 5.3 and 5.4 of the paper. The image includes the implementation of scalar interpolation algorithm, the PolyBench benchmark suite, and the scripts to run the benchmarks using auto-tuner and the cost model.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** A loop transformation to add scalar instructions into the vectorized code.
- **Program:** PolyBenchC-4.2.1 benchmark.

- **Compilation:** Includes LLVM 17.0.0.
- **Run-time environment:** Tested on Ubuntu 22.04 (6.8.0-48-generic) with Docker version 27.3.1.
- **Hardware:** Tested on Intel Xeon E5-2698 v4 CPUs.
- **Metrics:** Execution time.
- **Output:** Execution times in form of CSV files and the corresponding graphs for each experiment.
- **Experiments:** Cost model effectiveness (Section 5.3) and Scalar Interpolation performance (Section 5.4)
- **How much disk space required (approximately)?:** 1.9 GB for the docker image.
- **How much time is needed to complete experiments (approximately)?:** About 8 hours for a single run. It takes about 80 hours to run the experiments 10 times and replicate the paper experiments
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Ohio State University Software Distribution License, Apache License v2.0 with LLVM Exceptions.
- **Archived (provide DOI)?:** 10.5281/zenodo.14090974

A.3 Description

A.3.1 How to Access. The artifact is archived: 10.5281/zenodo.14090974 [13].

A.3.2 Hardware Dependencies. An Intel Xeon E5-2698 v4 CPU, because the cost model for the algorithm depends on specific hardware characteristics. The transformation still performs on other architectures, however the results may differ from the paper.

A.3.3 Software Dependencies. A working Docker installation.

A.4 Installation

A.4.1 CPU Frequency Scaling. To have consistent results, we need to lock the CPU frequency. In the experiments shown in the paper, it was locked at 2.2 GHz.

```
$ sudo cpupower frequency-set -g performance
$ sudo cpupower frequency-set -u 2.2GHz
$ sudo cpupower frequency-set -d 2.2GHz
```

A.4.2 Docker Setup. After downloading the compressed docker image file, load the image via the following command, and run the container:

```
$ docker load --input scalar-interpolation.tar.gz
$ docker run -it si:v1
```

A.5 Experiment Workflow

After running the container, you can set up the `config.json` file to change the experiment setting. The parameters of this file are explained in more detail in A.7.

```
$ nano config.json
```

The following commands run the experiments and generate the results.

```
$ python3 main.py
$ python3 main.py -a
```

The first command runs the experiments whose results are shown in Figures 2 and 3 of the paper, and the second command runs the experiment whose results are illustrated in Figure 5. An important note to replicate the results is that before running any of the above commands, the results from the previous runs must be removed from *root*/*results_file* path, which is where the results of the previous runs are stored. *root* and *results_file* are two keys in the config.json file that are used to set up the output path. The results of these experiments are stored in the *figures* and *tables* folders, to retrieve them, one should run the following commands:

```
$ ID=$(docker ps -q --filter 'ancestor=si:v1')
$ docker cp $ID:/home/si/figures/ .
$ docker cp $ID:/home/si/tables/ .
```

A.6 Evaluation and Expected Results

Using a similar environment and setup, the results should follow the same trend as shown in the paper. Yet, changing different factors, such as using a different CPU, or changing the number of repetitions may affect the final results.

A.7 Experiment Customization

After running the container, the experiment scripts can be set up using the `config.json` file. This file contains some file paths required to run the scripts and some parameters that can change the experiments, including:

- **repeat:** If it is set to N (N > 1), the experiments will be repeated N times and the scripts report average execution time.
- **si_factors:** list of candidate SIFs for the auto-tuner.
- **results_file:** a path to a file to save execution time and standard deviation results for the experiments.
- **measurement_retrials:** If it is set to N, it retries each measurement up to N times when the measurement fails for any reason.

A.8 Notes

- On the Experiment Workflow instructions, after running each command, the average execution time of each benchmark and the standard deviation are stored in a file whose path is set by **results_file** in config.json. This file must be removed before running each experiment replication command because it also serves as a caching mechanism in cases where the running experiment crashes.
- You may need sudo privileges when running docker depending on the host's configuration.

References

[1] 2008. *GCC Instruction Scheduling.* https://gcc.gnu.org/wiki/InstructionScheduling

[2] 2024. *llvm::ScheduleDAGInstrs Class Reference.* https://llvm.org/doxygen/classllvm_1_1ScheduleDAGInstrs.html

[3] 2024. *llvm::TargetTransformInfo Class Reference.* https://llvm.org/doxygen/classllvm_1_1TargetTransformInfo.html

[4] 2024. *llvm::VPRecipeBase Class Reference.* https://llvm.org/docs/doxygen/classllvm_1_1VPRecipeBase.html

[5] Saman Amarasinghe, Rodric Rabbah, and Samuel Larsen. 2009. Selective Vectorization for Short-Vector Instructions. (2009).

[6] David Bernstein and Michael Rodeh. 1991. Global instruction scheduling for superscalar machines. *SIGPLAN Not.* 26, 6 (may 1991), 241–255. https://doi.org/10.1145/113446.113466

[7] David Bernstein, Micheal Rodeh, and Izidor Gertner. 1989. Approximation algorithms for scheduling arithmetic expressions on pipelined machines. *J. Algorithms* 10, 1 (mar 1989), 120–139. https://doi.org/10.1016/0196-6774(89)90027-8

[8] S. Carr and Yiping Guan. 1997. Unroll-and-jam using uniformly generated sets. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. 349–357. https://doi.org/10.1109/MICRO.1997.645832

[9] Steve Carr and Ken Kennedy. 1994. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst.* 16, 6 (nov 1994), 1768–1810. https://doi.org/10.1145/197320.197366

[10] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All you need is superword-level parallelism: systematic control-flow vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 301–315. https://doi.org/10.1145/3519939.3523701

[11] Thomas J. Watson IBM Research Center. Research Division, FE Allen, and J Cocke. 1971. *A catalogue of optimizing transformations.*

[12] M Anton Ertl and Andreas Krall. 1991. Optimal instruction scheduling using constraint logic programming. In *International symposium on programming language implementation and logic programming*. Springer, 75–86.

[13] Reza Ghanbari, Henry Kao, Joao Paulo Labegalini de Carvalho, Ehsan Amiri, and Jose Nelson Amaral. 2024. Artifact of "Scalar Interpolation: A Better Balance Between Vector and Scalar Execution for SuperScalar Architectures". Zenodo. https://doi.org/10.5281/zenodo.14090974

[14] Philip B. Gibbons and Steven S. Muchnick. 1986. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction* (Palo Alto, California, USA) *(SIGPLAN '86)*. Association for Computing Machinery, New York, NY, USA, 11–16. https://doi.org/10.1145/12276.13312

[15] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.

[16] S. Larsen, R. Rabbah, and S. Amarasinghe. 2005. Exploiting vector parallelism in software pipelined loops. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. 11 pp.–129. https://doi.org/10.1109/MICRO.2005.20

[17] Hugh Leather, Edwin Bonilla, and Michael O'boyle. 2014. Automatic feature generation for machine learning–based optimising compilation. *ACM Trans. Archit. Code Optim.* 11, 1, Article 14 (feb 2014), 32 pages. https://doi.org/10.1145/2536688

[18] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. 2012. A compiler framework for extracting superword level parallelism. *SIGPLAN Not.* 47, 6 (jun 2012), 347–358. https://doi.org/10.1145/2345156.2254106

[19] Charith Mendis and Saman Amarasinghe. 2018. goSLP: globally optimized superword level parallelism framework. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 110 (oct 2018), 28 pages. https://doi.org/10.1145/3276480

[20] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: Padded SLP automatic vectorization. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 190–201. https://doi.org/10.1109/CGO.2015.7054199

[21] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. 2018. Look-ahead SLP: auto-vectorization in the presence of commutative operations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/3168807

[22] Louis-Noël Pouchet. 2018. *PolyBench/C - the Polyhedral Benchmark suite.* https://sourceforge.net/projects/polybench/

[23] Rodrigo C. O. Rocha, Vasileios Porpodas, Pavlos Petoumenos, Luís F. W. Góes, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Vectorization-aware loop unrolling with seed forwarding. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3377555.3377890

[24] Ira Rosen, D. Nuzman, and Ayal Zaks. 2007. Loop-aware SLP in GCC. (01 2007), 131–142.

[25] Vivek Sarkar. 2000. Optimized unrolling of nested loops. In *Proceedings of the 14th International Conference on Supercomputing* (Santa Fe, New Mexico, USA) *(ICS '00)*. Association for Computing Machinery, New York, NY, USA, 153–166. https://doi.org/10.1145/335231.335246

[26] J. Shin, M. Hall, and J. Chame. 2005. Superword-level parallelism in the presence of control flow. In *International Symposium on Code Generation and Optimization*. 165–175. https://doi.org/10.1109/CGO.2005.33

[27] Inderpreet Singh, Sunil K. Singh, Rashandeep Singh, and Sudhakar Kumar. 2022. Efficient Loop Unrolling Factor Prediction Algorithm using Machine Learning Models. In *2022 3rd International Conference for Emerging Technology (INCET)*. 1–8. https://doi.org/10.1109/INCET54531.2022.9825092

[28] M. Stephenson and S. Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *International Symposium on Code Generation and Optimization*. 123–134. https://doi.org/10.1109/CGO.2005.29

[29] Linda Torczon and Keith Cooper. 2007. *Engineering A Compiler* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[30] Kent Wilken, Jack Liu, and Mark Heffernan. 2000. Optimal instruction scheduling using integer programming. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/349299.349318

[31] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

Received 2024-09-12; accepted 2024-11-04