An Empirical Study of the Effect of Source-level Transformations on Compiler Stability

Zhangxiaowen Gong†, Zhi Chen*, Justin Josef Szaday†, David C. Wong‡, Zehra Sura §, Neftali Watkinson *, Saeed Maleki¶, David Padua †, Alexandru Nicolau *, Alexander V Veidenbaum *, Josep Torrellas†
‡ Intel Corporation, § IBM Research, ¶ Microsoft Research,
* University of California, Irvine, † University of Illinois at Urbana-Champaign,
Email: *{zhic2, nicolau, alexv, watkinso}@ics.uci.edu, ‡david.c.wong@intel.com,
†{gong15, szaday2, padua, torrella}@illinois.edu, §zsura@us.ibm.com, ¶saemal@microsoft.com

***Abstract*—Loop-level compiler optimizations are applied in a complex process with no guarantee that the code produced is optimal. Compilers also struggle to maintain a stable performance on different loops with the same semantics. This paper presents an analysis of the stability of the compilation process and shows potential for state-of-the-art compilers to improve code performance. In the study, loop nests are first extracted from benchmarks; then, sequences of source-level loop transformations are applied to these loop nests to create numerous semantically equivalent mutations; finally, the impact of transformations on code quality in terms of locality, dynamic instruction count, and vectorization is analyzed for different compilers. Our results show that up to 47% of the loops can be improved with at least a 1.15x speedup by this process while the average speedup can reach 1.6x for the improved loops. In addition, we propose a novel stability score that demonstrates the difference in stability from the studied compilers. The study concludes that the effect of source-level transformations varies among compilers, and the evaluated compilers have long ways to go until reaching stable.**

I. Introduction

Two of the most important outcomes after sixty years of compiler research are powerful methods for program analysis, and an extensive catalog of program transformations. Although there is room for improvement in these areas, we can say that the technology developed rests on solid ground and is well understood. On the other hand, the process of program optimization, which guides the application of transformations to achieve good performance, is not well understood. It is difficult to predict how an optimizing compiler and the options it presents a programmer will affect the quality of the generated code. This is why the standard way of selecting the best compiler command options is to use empirical methods (e.g. [1], [2]).

This paper aims to measure the effectiveness of the optimization pass of the three most popular compilers today: *GCC*, *ICC*, and *LLVM (Clang)*. It focuses on the compilation of `for` loops because this is the language construct whose analysis and transformation is best understood. We chose to focus on code performance headroom and stability, but other characteristics can also be evaluated using the same approach. We say that a compiler is stable if it produces the same target code for all versions of the source code resulting from semantic-preserving transformations. The performance headroom is a measure of the effectiveness of the optimization process. Although we can only obtain a lower bound of this headroom by trying limited combination of transformations, we expect this result, together with a figure of merit for stability, to be a useful approximation to the absolute performance headroom. And, by repeating the measurements along the years, these values could give us a measure of progress.

In this study, we used a wide variety of loop types obtained using an *extractor* that we developed and collected from 13 benchmarks suites and other sources, such as software libraries and machine learning kernels. The extractor separates `for` loop nests from the source code of the original applications and then builds standalone codelets that can be executed and measured. Only loop nests consuming more than 1,000 processor cycles were used in this study to limit the effect of measurement overhead. As a result, between 1,175 and 1,266 loop nests are investigated, depending on the compiler.

Because source-to-source transformations as a compiler pre-pass is a proven technique [3], [4] to study compiler effectiveness, we apply such transformations to obtain multiple semantically-equivalent versions of each loop for estimating headroom and stability. These versions are called *mutations* in this paper, and an automated tool we developed that generates them is called the *mutator*. Our source-to-source transformations are combinations of five basic yet most effective loop transformations: interchange, tiling, unrolling, unroll-and-jam, and distribution [5]. From the loop nests that we studied, a total of 64,928∼66,392 mutations are generated. The mutations are then compiled by each compiler we study. Because vectorization support is ubiquitous in modern processors and can have significant impact on performance, we also experimented with different vectorization settings during the compilation process. The execution times of the compilers' outputs are measured together with a number of hardware performance counters to help us better understand the effect of the transformations. The performance of each of the mutations is then compared to that of the original loop. Because the mutations are obtained by applying transformations that are widely used and can be easily implemented by any compiler, their effect is a good indication of whether or not there is room for improvement. Our paper demonstrates that even though these transformations are implemented by the compilers at hand, it is not enough, and that the application of the transformations in just the

right parameters can yield noticeable improvements in the performance of the resulting code. Also, the stability of a compiler is measured by how much the performance fluctuates among semantically equivalent mutations.

Our results show that there is a significant performance headroom for each of the three compilers evaluated. The application of source-to-source transformations as a pre-pass alone results in 25.9∼36.6% of the loops studied seeing a performance improvement of 15% or more. By further tuning vectorization settings, the numbers rise to 35.7∼46.5%, and a loop nest can expect a 1.61x∼1.65x speedup on average if we manage to find a beneficial mutation and/or better vectorization setting for it.

We then analyzed how each of the five elementary transformations applied by the mutator affects performance. We used hardware performance counters and manual inspection for each transformation to establish a correlation between the transformation and execution behavior in terms of locality, number of instructions executed, and vectorization. Three of the transformations resulted in interesting observations, we discuss their effect on specific loops to illustrate the complex ways in which they affect compiler outputs and challenges faced by compiler writers in developing stable optimization strategies. In addition, the effectiveness of the compilers' vectorizers is further evaluated by measuring the accuracy of their profitability model and investigating how source-level transformations affect the success rate and effectiveness of vectorization.

We quantified a compiler's degree of stability by introducing a *stability score* and found that the evaluated compilers are far from being stable. We also investigated if source-level transformations are able to narrow the performance gap among compilers and quantitatively confirmed it by devising a *convergence score*. Furthermore, our analysis indicates that as the number and complexity of the optimization passes in modern compilers increase, it becomes increasingly difficult to predict the impact of source-level transformations; thus, it becomes harder for programmers (and source-level optimizers such as polyhedral compilers) to control the behavior of their code by altering the loop structure, and the fact that different compilers may react to the same transformation variously makes it even more difficult.

The rest of the paper is organized as follows. In Section II, we describe in detail how we extract loop nests and generate mutations from them. Section III and Section IV present the compiler settings and quantitative results, respectively. In Section V, we analyze how different transformations affect performance. In Section VI, we further explore how vectorization setting impacts performance. Finally, Section VII discusses related work, and Section VIII presents our conclusions.

II. Loop Extraction and Mutation

Our study focuses on `for` loops written in C extracted from a variety of benchmark suites. We developed two components to study the impact of source-to-source transformations on the compilation of a loop nest: a loop nest *extractor* and a *mutator*, both based on the *ROSE* [6] source-to-source compiler infrastructure.

A. The Extractor

The extractor encapsulates loop nests from a benchmark application into individual standalone *codelets*. It first identifies `for` loops in benchmark source files by scanning the abstract syntax tree (AST). It then instruments each loop to save all input data to the loop. A loop nest may be executed multiple times, e.g. inside a function that is invoked multiple times. To save time, our system currently extracts data from only one of the loop executions chosen via the reservoir sampling algorithm. This algorithm grants each execution instance an equal chance to be selected [7].

Next, the extractor copies the source code of the loop nest to create a *codelet*. In a codelet, the loop nest is surrounded by operations that (I) initialize variables and memory regions with the captured data loaded from file, (II) record time via `RDTSCP` instruction, which allows accurate timing measurements, (III) read hardware performance counter values, (IV) use all the data written in the loop body to generate output value(s), so that the compilers do not remove operations as dead code, (V) repeatedly execute the loop nest for 100 times and record the median of the execution time. Variables and memory regions are reinitialized before each re-execution. The cache state is different from that of the original benchmark execution yet consistent among re-executions (except for the first execution). The codelets are thus completely self-contained and ready for compilation and execution.

B. The Mutator

The extracted loop nests, now as standalone executable codelets, are processed using a *ROSE*-based mutator to create semantically equivalent mutations. The mutator applies a sequence of source-to-source transformations to nests of `for` loops of the form

```
for(i=lb;i<=ub;i+=step).
```

The available transformations are interchange, tiling, unrolling, unroll-and-jam, and distribution. These five transformations are among the best-known loop transformations and relatively easy to implement. Therefore, they could be easily added to any compiler if they are not currently present. For each mutation, the mutator applies one or more of these transformations in sequence with various parameters.

Because the number of mutations of a single loop nest may grow exponentially as the number of transformation sequences and the selection of parameters to each transformation increase, we had to impose limitations to the transformation sequences so that the total number of mutations generated from the extensive collection of loop nests that we study remains reasonable.

First, the mutator does not explore the transformation space exhaustively. It instead transforms the loop nests in a selection of orders. The possible orders are subsets of:

$interchange \rightarrow unroll\text{-}and\text{-}jam \rightarrow distribution \rightarrow unrolling$
$interchange \rightarrow tiling \rightarrow distribution \rightarrow unrolling$

Thus, the maximum length of transformation sequence is 4. These orders are chosen such that the transformations that only operate on perfectly nested loops (all assignment statements are contained in the innermost loop), namely interchange, tiling, and unroll-and-jam are not applied after any transformation that may render the loop nests imperfect, namely distribution, unrolling, and unroll-and-jam.

Second, the parameters to each transformation are also limited. Table I shows the transformations and their parameters. For interchange, we explore every possible permutation, and the parameter for it is a number denoting the permutation in lexicographical order. For tiling, we tile a single dimension only, and parameters are the sizes used for strip mining plus the loop level that is strip-mined. For unrolling, we only unroll the innermost loop(s). If there are multiple loops at the innermost level, the mutator will unroll all of them by the same number of times. We use the unroll factor as parameter. For unroll-and-jam, we apply it at each non-innermost level, and the parameters are the loop level to be unrolled and the unroll factor. For distribution, we distribute statements in the innermost loop as much as possible based on the dependence analysis result; hence, it does not take parameters.

Transformation	**Parameters**	**Maximum # of variation**
Interchange	Lexicographical permutation number	$depth!$
Tiling	Loop level, tile size (8, 16, 32)	$depth \times 3$
Unrolling	Unroll factor (2, 4, 8)	3
Unroll-and-jam	Loop level, unroll factor (2, 4)	$depth \times 2$
Distribution	N/A	1

TABLE I: Transformations and their parameters

Although the above limitations confines the search space of transformation sequences to a manageable size for each loop nest, they also decrease the chance of finding the optimal transformation sequence for a loop nest. Therefore, in this study, we only aim to find lower bounds for performance headroom and instability of the investigated compilers.

In addition to imposed restrictions, the number of mutations is also limited by data dependence. Unrolling is the only transformation we use that is guaranteed to always be semantics-preserving. To ensure the legality of the other four, we use *PolyOptC* [8] for dependence analysis. The mutator applies only unrolling to loops that *PolyOptC* cannot analyze, such as loops with non-affine expressions in array subscripts or loops that cannot be expressed in polyhedral forms.

The depth and dependence graph of a loop nest determine how many mutations are generated from it. Among the loops we studied, up to 1,680 mutations were generated for a single loop nest. We extracted 3,197 codelets from various sources for this study, such as: benchmarks, audio/video codecs, deep learning libraries, and machine learning kernels. In total, we produced 100,219 mutations from these codelets; however, we only used the results from loops whose execution time exceeded 1,000 cycles. The final number of loops and mutations that we studied will be discussed in Section IV.

III. Compilers Evaluated

We used the loop nests and their mutations to evaluate recent versions of three widely used compilers: *GNU Compiler Collection (GCC)* 6.2.0, *Intel C++ Compiler (ICC)* 17.0.1, and *Clang* 4.0.0 (*LLVM*). The experiments were conducted on an Intel Xeon E5-1630 v3 processor (Haswell microarchitecture, 32KB private L1 instruction cache, 32KB private L1 data cache, 256KB private L2 cache, 10MB shared L3 cache) with 32GB DDR4 2133 RAM. The CPU has invariant TSC so that the readout from `RDTSCP` is accurate. To achieve stable results, all executions were fixed to the same core with dynamic frequency scaling, Intel *Hyper-Threading*, C-State, and *TurboBoost* technologies disabled.

When compiling the loop nests and their mutations, we turned on the following switches in addition to `-O3`. *GCC*: `-ffast-math` allows breaking strict IEEE compliance so that floating point operations can be reordered; `-funsafe-loop-optimizations` tells the loop optimizer to assume that loop indices do not overflow, and that loops with nontrivial exit condition are not infinite; `-ftree-loop-if-convert-stores` allows if-converting conditional jumps containing memory writes; *ICC*: `-restrict` and `-ipo` help with alias analysis; *Clang*: `-ffast-math` provides similar benefit as that in GCC; `-fslp-vectorize-aggressive` enables a second basic block vectorization phase. We instructed all three compilers to optimize for the native architecture, which supports vector extensions up to AVX2, and let the compilers' default vectorization profitability models determine when to vectorize loops.

IV. Results

This section presents the main results that we obtained. We report the performance of the evaluated compilers on the original loop nests in Section IV-A, present the general statistics of mutations' effects on performance in Section IV-B, demonstrate the statistics of each transformation's contribution in Section IV-C, discuss how various benchmarks react to mutations differently in Section IV-D, and finally propose metrics to measure compiler stability and convergence effect in Section IV-E and IV-F respectively.

A. Baseline Performance

To minimize the impact of timing noise, we only examine loops with an execution time of at least 1,000 cycles. Consequently, depending on the compiler, between 1,175 and 1,266 loops are included in the study. Table II lists the number of loops and their mutations from different benchmarks that are included for each compiler.

To compare the compilers' performance on the baseline loops, we only consider the 1061 loops that are shared by all three compilers' results. On average, GCC and Clang generate code that is 1.06x and 1.27x slower respectively than ICC;

Benchmark	# of loops (# of mutations)		
	GCC	ICC	Clang
ALPBench[9]	24 (72)	22 (66)	31 (129)
ASC-llnl[10]	22 (350)	21 (347)	22 (350)
Cortexsuite[11]	60 (1060)	57 (791)	62 (1042)
FreeBench[12]	38 (242)	31 (141)	39 (245)
Intel PRK[13]	36 (286)	23 (189)	34 (261)
Livermore[14]	53 (1443)	51 (1436)	57 (1612)
MediaBench[15]	152 (773)	120 (532)	183 (1279)
Netlib[16]	25 (207)	21 (195)	24 (204)
NPB[17]	196 (52259)	195 (52244)	198 (52350)
Polybench[18]	90 (3574)	91 (3589)	91 (3589)
SPEC 2000[19]	122 (1263)	125 (1272)	129 (1337)
SPEC 2006[19]	102 (421)	103 (425)	129 (907)
TSVC[20]	149 (1955)	149 (1955)	149 (1943)
ML kernels[21]	27 (177)	27 (177)	21 (123)
Libraries[22], [23], [24], [25], [26]	145 (1735)	139 (1569)	97 (1023)
Subtotal	1241 (65817)	1175 (64928)	1266 (66392)
Shared	1061 (63902)		

TABLE II: The numbers of loop nests and their mutations included in the study

however, they generate code that outperforms ICC's by 15% in 174 and 114 cases respectively (this threshold was chosen to mitigate experimental timing noise). In general, the results indicate that there exist types of loops that each compiler can handle better than the others.

B. General Mutation Performance

If we average the speedup of each loop's best mutation over its baseline, we see that, on average, source-level transformations speed up the loops by 1.11x, 1.05x, and 1.16x for GCC, ICC, and Clang respectively, as shown in the second row in Table III. Also, the standard deviations of speedup are 1.02∼1.04, suggesting that the range of speedup is significant.

	GCC	ICC	Clang
# of loops studied (L)	1241	1175	1266
μ_g (σ_g) of best mutation to baseline speedup	1.11 (1.02)	1.05 (1.04)	1.16 (1.03)
# (%) in L that have beneficial mutation(s)	402 (32.4%)	304 (25.9%)	463 (36.6%)
# (%) in L that have all mutations unfavorable	89 (7.2%)	188 (16.0%)	73 (5.8%)

TABLE III: Number of loops with mutation speedup above/-below thresholds

Next, we assign categories to each of the mutations based on their impact on performance. We consider mutations that generate code 15% faster than the baseline to be *beneficial* and those that generate code that is 15% slower than the baseline to be *unfavorable* mutation, the rest are considered to be *neutral*.

As shown in the second row of Table III, the percentage of loops with at least one beneficial mutation ranges from 25.9% (ICC) to 36.6% (Clang). This suggests that Clang benefits more from source-level transformations than ICC, with GCC sitting somewhere between the two. On the other hand, as shown in the last row of Table III, the percentage of loops that only have unfavorable mutations is much higher for ICC (at 16.0% vs. 5.8% (Clang) and 7.2% (GCC)), implying that the performance of code generated by ICC is more easily worsened by source-level transformations compared to Clang and GCC.

Focusing on loops with beneficial mutations, we get the distribution of speedups shown in Figure 1. The plots reveals that for all three compilers, although most of the speedups are below 2x, a significant number of loops receive over a 2x speedup. In fact, there are loops with speedups as high as 20x; however, we found that most speedups over 6x are due to pathological scalar optimization after unrolling. For example, after the mutator unrolls a loop from *TSVC* by 8 times, Clang decides to further fully unroll the loop and pre-calculates most of the scalar operations at compile time, speeding up the loop by 20x. Nevertheless, there is a case where interchange facilitated better locality and vectorization to help a loop nest gain 15x performance with Clang.

While ICC has fewer loops that are sped up by mutations, the number of loops that have over a 3x speedup is comparable to Clang's and is much greater than GCC's. Furthermore, both ICC and Clang obtain 1.54x speedup if there is a beneficial mutation for a loop whereas GCC can only attain a 1.46x speedup. This suggests that GCC often has less dramatic performance improvement compared to Clang and ICC.

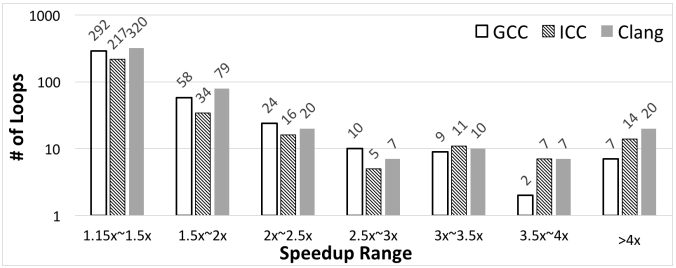


Fig. 1: Distribution of loops with beneficial mutation(s) and their speedup

Next we consider loops where all mutations are unfavorable. Figure 2 shows the distribution of loops at various slowdown range. The average slowdown for these cases are 1.56x, 1.54x, and 1.49x for ICC, GCC, and Clang respectively. It turns out that most slowdowns are less than 2x but, for several loops, slowdowns are greater than 4x and may be as large as 14x in extreme cases. Considering that every loop has a mutation as simple as unrolling by two, these results are surprising. After examining the extreme cases, we learned that large slowdowns are often tied to a sharp increase in instruction count, implying that the compilers generate inefficient code when faced with harmful mutations.

Table IV lists the statistics of the length of transformation sequence that produces the best beneficial mutation for a loop. For all three compilers, over 75% of the best mutations only have one transformation, and because the effect of transformation sequence is usually the combination of the effects of the individual transformations in the sequence, in later sections we will mainly discuss single transformations.

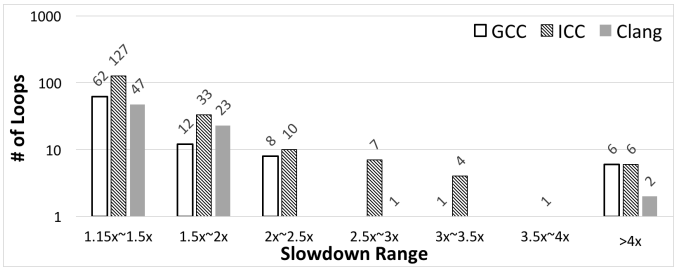


Fig. 2: Distribution of loops with all mutations unfavorable in different slowdown range

Transformation sequence length	# (%) in the best beneficial mutations		
	GCC	ICC	Clang
1	316 (78.6%)	228 (75.0%)	351 (75.8%)
2	62 (15.4%)	55 (18.1%)	101 (21.8%)
3	24 (6.0%)	21 (6.9%)	11 (2.4%)
4	0 (0.0%)	0 (0.0%)	0 (0.0%)

TABLE IV: Statistics of the length of transformation sequence that produces the best beneficial mutation for a loop

C. Effects of Different Transformations

Table V contains the statistics of speedup gained by different elementary transformations. The values are computed from mutations that undergo only the respective transformation. In each cell, the first two numbers are the geometric mean and standard deviation of speedup, and the last one is the percentage of affected mutations that are beneficial. In general, all three compilers react to the same transformation similarly with some exceptions. While GCC and Clang on average receive a speedup from unrolling, ICC expects a slowdown. Also, distribution is able to help ICC much more than for the other two.

Transformation	μ_g/σ_g of speedup/% being beneficial		
	GCC	ICC	Clang
Interchange	0.66/1.06/9.0%	0.69/1.05/6.4%	0.63/1.07/9.0%
Tiling	0.83/1.03/9.5%	0.91/1.03/9.2%	0.94/1.05/16.2%
Unrolling	1.06/1.03/25.8%	0.97/1.04/18.0%	1.09/1.05/29.6%
Unroll & jam	1.01/1.02/22.7%	1.02/1.06/16.2%	1.10/1.04/32.4%
Distribution	1.12/1.06/27.9%	1.25/1.11/34.0%	1.05/1.04/27.0%

TABLE V: Statistics of speedup from different elementary transformations

While unrolling, unroll-and-jam, and distribution on average increase performance, the two locality focusing transformations, interchange and tiling, are expected to slowdown the loop. The intuitive reason is that most of the original loops are already written with locality in mind, so altering locality will likely lead to sub-par results. Nonetheless, 6.4%∼9.0% of the interchanged mutations and 9.2%∼16.2% of the tiled mutations manage to be beneficial.

D. Effects of Mutations on Different Benchmarks

Figure 3 shows the average speedup achieved by the best mutations over their baseline grouped by benchmark-compiler combinations.

Different compilers receive various effects even on the same benchmark. In general, the best mutations can on average speed up GCC and Clang for most of the benchmarks except for *ALPBench*, *Intel PRK*, and the machine learning kernels, where the average speedup is close to or slightly lower than 1x. On the other hand, there is usually less room to improve the performance of loop nests for GCC across all benchmarks. This is mainly because ICC is more aggressive in optimization compared to the other compilers. It affects the results in two ways, (I) heavy optimization may leave less room for improvement, (II) ICC may sometimes revoke source-level transformations. There are cases where it re-rolls or permutates the loop back after we apply unrolling or interchange on a loop.

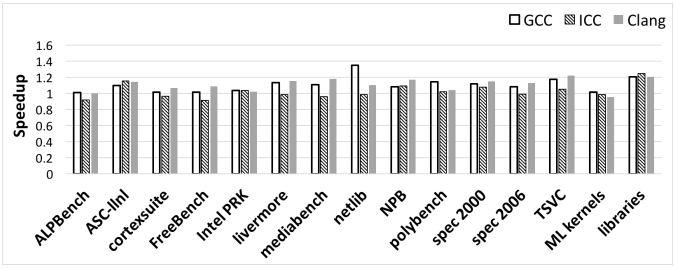


Fig. 3: Average speedup that the best mutations achieve from different benchmarks

The average speedup surpasses 1.15x for the following benchmark-compiler combinations: *GCC* on netlib (1.35x), TSVC (1.18x) and libraries (1.25x); *ICC* on ASC-llnl (1.15x) and libraries (1.21x); *Clang* on livermore (1.15x), mediabench (1.18x), NPB (1.17x), TSVC (1.22x), and libraries (1.21x).

E. Compiler Stability

We would expect a perfect compiler to be able to undo unfavorable transformations and apply beneficial transformations to any mutation of a loop. We would call such a compiler *stable*, since it would produce the same performance for any mutation of a given loop. We quantify the stability of a compiler as a stability score, as given by,

$$\frac{1}{L}\sum_{l=1}^{L}\frac{\sigma(t^{(l)}_{baseline}, t^{(l)}_{mutation[0]}, \ldots, t^{(l)}_{mutation[n^{(l)}]})}{\mu(t^{(l)}_{baseline_l}, t^{(l)}_{mutation[0]}, \ldots, t^{(l)}_{mutation[n^{(l)}]})} \quad (1)$$

This is essentially the average stability of L loops. We compute the stability of an individual loop as the normalized standard deviation of the execution times of its n mutations and its baseline. Standard deviation was chosen since it would be close to 0 if a compiler produced stable performance for a loop. The stability score has no absolute meaning by itself. Instead, it can be used to compare the relative stability among different compilers or to track the change in stability of a compiler over its different versions.

We believe a compiler's stability score reflects its ability to recognize optimization opportunities. For example, a stable compiler would interchange an unfavorably interchanged loop

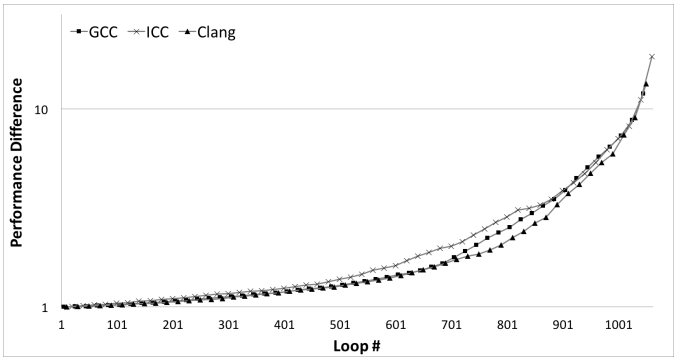


Fig. 4: Performance difference between each loop's best-to-worst mutations

for better memory access patterns and/or vectorization opportunities. The scores are 0.233, 0.201, and 0.183 for ICC, GCC, and Clang respectively. By definition, a larger stability score reflects greater instability; therefore, among the compilers we tested, ICC is the most unstable and Clang is the most stable, with GCC somewhere in between them.

To demonstrate the instability, we plot the performance difference from each loop's best mutation to its worst one in log scale as shown in Figure 4. The performance differences are taken from the 1,061 loops that are shared by all three compilers and then sorted for each compiler separately; thus, the data points at the same x location do not necessarily represent the same loop. The plot clearly shows that ICC typically has the highest performance difference while Clang has the lowest.

Our results show that all three compilers have significant best-to-worst performance difference for most of the loops, and the difference can be as high as 29x. Therefore, the compilers still have a long way to go before being stable.

F. Convergence Effect on Performance

In Section IV-A, we mentioned that the three compilers sometimes have better baseline performance compared to the others. Source-level transformations may help the compilers that are falling behind catch up. To quantify this, we propose a *convergence score* to measure how close the transformations bring the performance of the compilers to one another, as given by,

$$\frac{1}{L}\sum_{l=1}^{L}\frac{\sigma(t_{GCC}^{(l)}, t_{ICC}^{(l)}, t_{Clang}^{(l)})}{\mu(t_{GCC}^{(l)}, t_{ICC}^{(l)}, t_{Clang}^{(l)})} \tag{2}$$

Like the stability score, the convergence score is an average of normalized standard deviations. This time, the standard deviation measures how close the compilers' performances are. The convergence score is also meaningful only when used in comparison. We computed the baseline convergence score as 1.39 and as 0.96 when comparing the best mutations. We believe that the smaller convergence score for the best mutations shows that applying source-level transformations mitigates deficiencies of compilers, tightening the performance gap between them.

V. Effect of Transformations

In this section, we look at the results in more detail. We investigate how each of the five transformations applied by the mutator impact performance by computing the correlation coefficient between performance change and each of a number of hardware performance counter readouts. We also present a few case studies illustrating the complexity of interaction between the transformations applied by the mutator and the compilers.

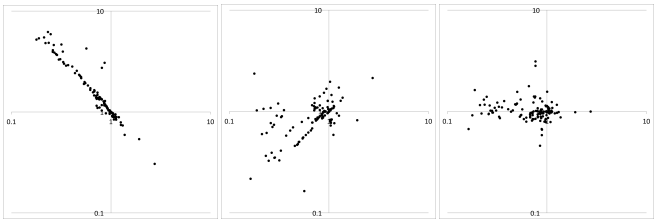
A. Computing correlation coefficients

We compute the correlation coefficient between (a) the change in value of a performance metric and (b) the change in execution time. Here "change" means the difference between the original loop and the transformed loop. A performance metric could be obtained by reading a hardware performance counter or it could be computed using values from hardware counters as is the case for cache miss rate. Specifically, the correlation coefficient for a transformation T is computed as follows: (I) For each loop nest l, we determine the fastest mutation, m, resulting from applying the transformation. Recall that transformations can produce multiple mutations since they are controlled by parameters (Table I). (II) Compute the ratio of the execution time the original loop nest and the execution time of the mutation m ($1/S = t_m/t_{original}$). This is the inverse of the speedup of m over the original loop. (III) Compute the ratio of the values for a performance metric P for the original loops and the mutation m ($R = P_{original}/P_m$); (IV) Calculate the Pearson correlation coefficient [27] between the performance ratio $1/S$ and the metric ratio R, denoted as $\rho_{1/S,R}$ for all loop nests that can be transformed by T. The values obtained from the metric are inaccurate since the change in their value during the reading process is also included in the final readout. This inaccuracy is higher for loops with short execution times. Hence, for this analysis, we only include loops with baseline execution time higher than 10,000 cycles.

Note that $\rho_{1/S,R} \in [-1, 1]$, where -1, 1, and 0 represent a perfect negative correlation, a perfect positive correlation, and no correlation, respectively. When the absolute value $|\rho_{1/S,R}|$ is high for a metric P, we may expect transformation T to affect performance mainly in a way that relates to the factors measured by P. If, on the other hand, $|\rho_{1/S,R}|$ is not large for any of the performance metrics, the transformation may impact performance for multiple reasons and thus its effect is less clear.

To illustrate these ideas, consider Figure 5, which plots the ratio of performance factor values R (y-axis) against speedup S (x-axis) in logarithmic scale. For a perfect correlation ($\rho_{1/S,R} \in \{-1, 1\}$), all points on the plot are expected to be on a $R = kS, k \neq 0$ line. Figure 5 (a) is a plot with high negative $\rho_{1/S,R}$ value (-0.88), and the points resemble a line $R = kS$ with $k < 0$. Figure 5 (b), on the other hand, presents a mid-range positive correlation $\rho_{1/S,R} = 0.31$. We can still see a distinguishable $R = kS$ with $k > 0$, but the points are more scattered. Finally, Figure 5 (c) plots a $S - R$

relationship with a close to 0 $\rho_{1/S,R}$. In this case, the figure does not manifest a visual correlation. In the rest of the section, we consider $|\rho_{1/S,R}| \in [0.2, 0.5)$ as moderate correlation, and $|\rho_{1/S,R}| \in [0.5, 1]$ as high correlation.



(a) $\rho_{1/S,R} = -0.88$ (b) $\rho_{1/S,R} = 0.31$ (c) $\rho_{1/S,R} = -0.01$

Fig. 5: Example visualization of different $\rho_{1/S,R}$ range

The full list of performance metrics used in our study contains 59 entries. For clarity, we only present the descriptions of performance metrics that show interesting correlations in the following sections in Table VI. In the table, derived metrics are highlighted in italic font.

Metric	Description
inst_retired.any	Counts the number of instructions retired from execution.
l1d.replacement	Counts when new data lines are brought into the L1 Data cache, which cause other lines to be evicted from the cache.
l1d_pend_miss.pending	Increments the number of outstanding L1D misses every cycle.
l1d_pend_miss.pending _cycles	Cycles with L1D load Misses outstanding.
mem_load_uops_retired .l1_miss_ps	Counts retired load uops in which data sources missed in the L1 cache.
l2_demand_rqsts.wb_hit	Not rejected writebacks that hit L2 cache.
l2_trans.l2_wb	L2 writebacks that access L2 cache.
l2_lines_in.all	Counts the number of L2 cache lines brought into the L2 cache.
l2_rqsts.miss	All requests that missed L2.
dtlb_load_misses .miss_causes_a_walk	Misses in all TLB levels that cause a page walk of any page size.
dtlb_load_misses.stlb_hit	Number of cache load STLB hits. No page walk.
%l1/l2/l3_hit/miss	L1/L2/L3 hit/miss rate
inst_rate	Instruction rate that measures instruction level parallelism
l2_rw_rate	L2 read/write rate calculate by dividing the total number of L2 requests by the cycle count

TABLE VI: Top performance metrics that correlate to execution time

B. Interchange

Loop interchange is useful when a nested loop initially has a non-unit-stride memory access pattern. If interchanging the loop nest can result in unit-stride access (or just reduce the size of the stride), performance may be improved due to better locality.

As expected, the correlation values obtained indicate that interchange is correlated with cache performance metrics for all three compilers. For ICC and Clang, we found high negative correlations ($\rho_{1/S,R} < -0.9$) with performance metrics that reflect change in L1 performance (`l1d.replacement`, `l1d_pend_miss.pending_cycles`, `l1d_pend_miss.pending`), as well as in L2 performance (`l2_trans.l2_wb`, `l2_lines_in.all`), `l2_demand_rqsts.wb`. Besides, for ICC there is a high negative correlation with L3 performance metric `%l3_miss`. For GCC, we also see moderate to high negative correlations (-0.4∼-0.7) with these cache related metrics. Moreover, interchange also affects TLB performance with -0.6∼-0.7 negative correlations with TLB related metrics (`dtlb_load_misses.stlb_hit`), `dtlb_load_misses.miss_causes_a_walk`. The locality improvements subsequently contribute to an increase in instruction level parallelism suggested by moderate positive correlations (0.2∼0.5) with `inst_rate`.

In addition, the performance of both GCC and Clang appears to be influenced by the change in dynamic instruction count (`inst_retired.any`) with $|\rho_{1/S,R}|$ ranging from $0.4 \sim 0.6$. This phenomenon implies that interchange can help these two compilers accomplish the same amount of work with fewer instructions. For ICC, we also found that performance change is often accompanied by change in instruction count in the opposite direction. By observing the instruction mix of affected loops, we corroborated that vectorization contributes to the reduction of instruction count.

```
for(k = 0; k < 25; k++) {
  for(i = 0; i < 25; i++) {
    for(j = 0; j < Inner_loops; j++) {
      Px[j][i] += Vy[k][i] * Cx[j][k];
}}}
/* interchanged */
for(j = 0; j < Inner_loops; j++) {
  for(k = 0; k < 25; k++) {
    for(i = 0; i < 25; i++) {
      Px[j][i] += Vy[k][i] * Cx[j][k];
}}}
```

Listing 1: Original and interchanged *Livermore Loops* code

Listing 1 is a case from the *Livermore Loops* that demonstrates how interchange affects instruction count differently in each of the three compilers. ICC gives the best performance for the original loop nest; Clang's output is 1.13x slower than ICC and GCC has a 1.45x slowdown. Based on manual inspection, we conclude that neither GCC nor Clang vectorize the loop due to a non-unit stride access. However, Clang manages to generate a more efficient address calculation than GCC, so Clang's scalar code executes 268K instructions while GCC executes 384K instructions. On the other hand, ICC vectorizes the loop using gather-scatter and its output executes 288K instructions. Therefore, the non-unit stride and gather-scatter overhead neutralizes great speedup from vectorization.

On the other hand, after the mutator interchanges the loop nest as shown in Listing 1, the accesses to `Px` and `Vy` become unit-stride, and the locality when reading `Cx` also improves. As a result, all three compilers vectorize the mutation with reduced gather-scatter effort. ICC's output now executes only 62K instruction, and is 2.8x faster than its baseline; GCC's numbers are 153K/2.4x, and Clang's are 33K/4.7x. Hence,

apart from affecting locality, interchange can enable and/or increase the effectiveness of vectorization. An additional note is that after interchange, Clang's mutation takes the lead and becomes 1.46x faster than ICC's output and 2.39x faster than GCC's. It is interesting to see a source-level transformation changing the outcome for which compiler has the best performance on a loop nest.

When scrutinizing the assembly, we noticed that the compilers do apply loop interchange in some cases. This means that the main reason for why these compilers fail to do interchnage properly, must be an inaccurate profitability model.

C. Unrolling

Loop unrolling is a technique traditionally used for better performance in exchange of bigger space. With the cost of duplicating the loop body, unrolling may improve performance by reducing control overhead (e.g. advancing the iterator and testing exit conditions) and/or by enabling scalar optimizations (e.g. common sub-expression elimination, constant folding, etc). It may also allow the compiler to perform instruction scheduling in a more flexible manner, thus increasing instruction level parallelism (ILP). Because of the trade-off between space and performance, compilers may decide whether to apply it after estimating potential benefit.

The correlation results demonstrate that unrolling indeed reduces instruction count. All three compilers show moderate to high negative correlations ($0.3 \sim 0.7$) with `inst_retired.any`. We also see moderate positive correlations ($0.3 \sim 0.4$) with `l2_rw_rate`, suggesting that although unrolling does not effectively reduce L1 miss rate, its ability to reduce instruction count and increase ILP can still increase performance by better utilizing L2 throughput when the locality can be captured at the L2 level.

By further examining the assembly code, we first noticed that the compilers do unroll some loops themselves. Sometimes they may even fully unroll a loop when the trip count is relatively low, potentially exploding the code size. However, in many cases when unrolling is beneficial, the compilers fail to apply it. In particular, we found that unrolling is considerably more effective towards loops whose bodies are small. The reason is that such loops suffer more from control overhead. Also, since the operation counts are low, unrolling does not excessively inflate the code size. It is surprising that the compilers decide not to unroll these loops, and just as with interchange, the compilers need better profitability models for unrolling.

Also, unrolling sometimes facilitates vectorization. Traditionally, vectorization is done by strip-mining the inner loop by the vector length and then replacing the original loop body with a vector equivalent. With this method, fully unrolled loops cannot be vectorized and partially unrolled loops could lead to inefficient vectorization. We found cases where unrolling disabled vectorization. For example, ICC and Clang are both able to vectorize a loop from *SPEC 2006* by default, but unrolling them twice causes the compilers to only produce scalar code, which leads to 12x and 14x slowdown respectively. Previous research suggests that compilers may even re-roll a source-level unrolled loop in order to vectorize it [28]. However, the more recent basic block vectorization technique incorporates unrolling in its process. This method first unrolls the loop by a factor and then tries to assemble isomorphic statements (which contain the same operations in the same order) in the unrolled loop body into vector instructions. Such vectorization is also referred to as superword-level parallelism (SLP) [29].

```
for (i = 0; i < 32000; i++) {
  x = a[32000 - i - 1] + b[i] * c[i];
  a[i] = x - 1.0;
  b[i] = x;
}
/* unrolled 8 times */
for (i = 0; i < 32000; i += 8) {
  x = a[32000-i-1] + b[i] * c[i];
  a[i] = x - 1.0;
  b[i] = x;
  /* iteration 2~7 are omitted */
  x = a[32000-(i+7)-1] + b[i+7] * c[i+7];
  a[i+7] = x - 1.0;
  b[i+7] = x;
}
```

Listing 2: Original and unrolled *TSVC s281* code

A benefit of basic block vectorization over the traditional method is that the former can partially vectorize a loop with ease. Although we witnessed numerous cases where unrolling helps the compiler to fully vectorize the loop, we present the case of loop *s281* from *TSVC*, as shown in Listing 2, because it exhibits the interesting trait of partial parallelization. This loop is not well-optimized by any of the considered compilers. After unrolling the loop 8 times, Clang first creates two temporary vectors, denoted as `tmp[0:7]` and `x[0:7]`. The former vector holds the intermediate results from 8 instances of sub-expression `b[i]*c[i]` while the latter vector is an extension of scalar `x`. Then, `tmp[0:7]=b[i:i+7]*c[i:i+7]` and `b[i:i+7]=x[0:7]` are vectorized because they do not have loop carried dependence. Since operations on `a[]` have loop carried dependences, they remain scalar. By partially vectorizing this loop, Clang gains a 1.7x speedup. The case demonstrates that although the compilers have implemented basic block vectorization, they may miss the opportunity until the loops are manually unrolled.

Because unrolling can have diverse impact on vectorization, it may be hard for programmers to predict its effect on performance.

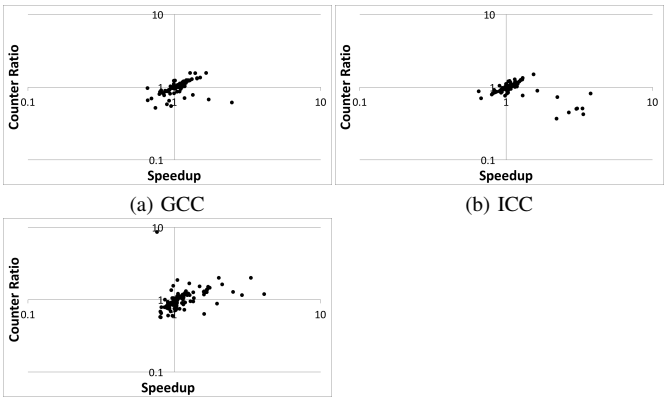
D. Unroll-and-jam

Unroll-and-jam is primarily employed to facilitate data reuse via improving register usage, therefore decreasing the number of memory access. Another notable utility of unroll-and-jam is that it may enable vectorization on the outer loop without performing interchange [30]. Although not as important, unroll-and-jam can also reduce control overhead similar to unrolling's effect.

The correlation results for unroll-and-jam are rather interesting. ICC has a high negative correlation (-0.9) with `l1d.replacement` and has a moderate positive correlation (0.4) with `%l1_hit`, which indicate that improvement in L1

hit rate is a major factor to the performance gain. GCC and ICC, on the other hand, have lower negative correlations ($0.5 \sim 0.6$) with `l1d.replacement` yet both also show moderate correlations ($0.5 \sim 0.6$) with `inst_retired.any`. The main difference between ICC and GCC/Clang is the correlations with `l2_rw_rate`. For this metric, ICC exhibits a high negative correlation (0.7) while GCC and Clang both have moderate positive correlation (0.3). These values imply that unroll-and-jam has a different effect on ICC compared to GCC/Clang.

Figure 6 contains plots of speedup (x-axis) vs. change in `l2_rw_rate` (y-axis) for all three compilers under the influence of unroll-and-jam. From the figure, we see that at low speedup/slowdown, all three compilers show positive correlations with the metric. However, for high speedup cases, ICC shows decrease in `l2_rw_rate`, represented by the points on the lower right. Since the Pearson correlation is biased towards data points with higher values, the overall correlation becomes negative. In order to understand ICC's diverse correlation with `l2_rw_rate` at different speedup ranges, we inspected other metrics of positive/negative correlation groups separately. For the negative correlation group, we found that `l1d.replacement` is significantly reduced, which means that the high speedup is achieved mainly from better data reuse and thus fewer L1 eviction. For the positive correlation group, we discovered that the relatively low speedup is due to other factors such as lower control overhead.



(a) GCC (b) ICC

(c) Clang

Fig. 6: Speedup vs. change in `l2_rw_rate` due to unroll-and-jam

On the contrary, GCC and Clang have fewer points in the lower right and more points showing positive correlation. More notably, Clang is able to obtain high speedup with a positive correlation with `l2_rw_rate`. By digging into other metrics, we learned that a majority of the speedup corresponds to the negative correlation with `inst_retired.any`, and the source of the reduction in dynamic instruction count is largely related to vectorization. For GCC, in the 42 cases where unroll-and-jam is beneficial ($>$ 1.15x speedup), 12 loops are not vectorized initially, and 4 of them become vectorized after unroll-and-jam. Also, we see a general increase in the vectorization rate ($\frac{vector\ instruction\ count}{total\ instruction\ count}$). From the 30 loops that are already vectorized at the beginning, 21 loops receive at least a 15% vectorization rate increase. For Clang, in the 55 cases where unroll-and-jam is beneficial, 9 loops are not vectorized initially, and 4 of them become vectorized afterwards. Note that these 4 loops contain the top 2 speedup that Clang attains through unroll-and-jam, achieving 4.1x and 3.6x respectively. Unroll-and-jam also help increasing the vectorization rate for 8 loops.

We also surprisingly discovered that while unroll-and-jam helps ICC's performance, it seemingly reduces the effectiveness of vectorization. In the 28 cases where unroll-and-jam is beneficial, 4 loops are not vectorized initially, and none of them gets vectorized after the transformation. Instead, there exist 5 loops whose baselines are vectorized yet the transformed mutations become not; nonetheless, unroll-and-jam manages to speedup these loops by 2.2x to 3.3x. Furthermore, unroll-and-jam reduces the vectorization rate of 7 loops by at least 15%. These results seem counter-intuitive at first, but after further investigation, we found 2 major factors that contribute to the anomaly. First, the benefit from vectorization is shadowed by a worse memory access pattern. For the cases where scalar mutations outperform vectorized baselines, we always see sharp reduction in L1 miss rate after unroll-and-jam. Second, the vectorized baseline may contain performance unfriendly patterns, such as gather-scatter, and unroll-jam helps generate more efficient vector code, even with a lower vectorization rate.

```
for(i = 0; i < n; i++) {
  for(j = 0; j < n; j++) {
    x[i] = x[i] + beta * A[j][i] * y[j];
  }
}
/* unroll-and-jammed 4 times*/
for(i = 0; i < n - fringe; i += 4) {
  for(j = 0; j < n; j++) {
    x[i] = x[i] + beta * A[j][i] * y[j];
    x[i+1] = x[i+1] + beta * A[j][i+1] * y[j];
    x[i+2] = x[i+2] + beta * A[j][i+2] * y[j];
    x[i+3] = x[i+3] + beta * A[j][i+3] * y[j];
  }
  /* residue loop is omitted */
}
```

Listing 3: Original and unroll-and-jammed *Polybench linear-algebra-blas-gemver* code

Listing 3 contains a loop nest from *Polybench*'s workload *linear-algebra-blas-gemver* that illustrates how unroll-and-jam helps ICC's vectorizer with a deceiving reduction in vectorization rate. ICC manages to vectorize the original loop nest, yet in an inefficient means. It first unrolls the inner loop by 32 times. It then transforms the unrolled iterations into 8 vector operation sessions. In each session, 4 elements of `A` are gathered from far apart addresses to assemble a vector, and another vector of 4 `y` elements are directly loaded from consecutive `y`. Then, the two vectors together with a third vector of copies of `beta` are multiplied together. The result vectors from all 8 session are later added together and reduced to a single value that is stored to `x[i]` afterwards. This vectorization is very inefficient in terms of both gather-scatter overhead and locality.

Fortunately, unroll-and-jam provides a better vectorization approach. ICC is able to vectorize the inner loop body with the basic block vectorization technique after the transformation. It first broadcasts `y[j]` to a 256-bit vector register. Then, it loads another vector register with `A[j][i:i+3]` from consecutive addresses. Afterwards, it multiplies the two vectors with a vector of `beta`. Each iteration the result of the multiplication is accumulated onto that from the previous iteration, and after the inner loop exits, the results are written to `x[i:i+3]` with a vector store instruction. Clearly, the new approach is superior since it completely eliminates gather-scatter and has improved locality. In the end, the mutation is 3.7x faster than the baseline. However, the mutation contains 46% vector instructions, whereas the baseline contains 64%, so in this case the vectorization rate is misleading.

E. Tiling

Tiling is applied to a loop nest primarily when the workset is reused multiple times but is too large to fit in cache. By tiling the loop nest with appropriate block size, blocks of the original workset can be reused without re-fetching from the lower memory, therefore improving performance.

The correlation results do confirm that tiling mainly affects locality. All three compilers exhibit different amount of correlations ($0.3 \sim 0.6$) with metrics related to various cache levels, such as L1 (`l1d_pend_miss.pending_cycles`, `%l1_hit`, `l1d_pend_miss.pending`, `l1d.replacement`), L2 (`l2_rqsts.miss`, `l2_lines_in.all`), L3 (`%l3_hit`), and TLB (`dtlb_load_misses.stlb_hit`).

The hardware counter values also suggest that tiling often increases dynamic instruction count, which mainly results from additional address calculations and iterator operations introduced by the added loop nest level. Therefore, when the access patterns are not benefit from tiling, or the benefit does not cover all the overhead, tiling can cause slowdown.

F. Distribution

Loop distribution helps performance mainly by separating data streams, which may improve locality and/or prefetching behavior. The correlation results confirm its utility. All three compilers have high positive correlations ($0.7 \sim 0.9$) with `l2_rw_rate`, indicating distribution may better utilize L2 throughput, likely because of better prefetching behavior. We also see moderate positive correlations ($0.3 \sim 0.5$) with `inst_rate`, which implies that higher L2 access throughput also helps increasing the overall instruction throughput.

VI. Vectorization

In Section V, we already discussed that vectorization plays a major role in the performance gain. If a loop is not originally vectorizable but, after undergoing a transformation sequence, becomes vectorized, it may receive a sizable performance boost. We also noticed that there are scenarios where scalar code outperforms vector code due to the overhead introduced during the vectorization process and/or locality difference. Therefore, we took one step further to investigate how different vectorization settings may influence the performance of loops.

We compiled and profiled all the loop nests and their mutations with 4 additional vectorization settings. We refer to the compiler settings described in Section III as the reference settings, and the 4 additional settings are the reference settings with added switches. The settings are: only generating scalar code, using only SSE, using SSE and AVX, and using up to AVX2. Moreover, for the three vector configurations, we disabled the compilers' vectorization profitability analysis if possible so that the compilers vectorize the loop with the corresponding vector extension whenever possible, regardless of the predicted profitability. Note that Clang does not provide switches to turn off the profitability analysis.

Because vectorization does not always guarantee speedup, instead of looking at compilers' vectorization report to determine whether a loop is vectorized, we define that a loop has *effective* vectorization if the vector code is at least 15% faster than the scalar code; specifically, we claim the vectorization is effective if $t_{scalar} / \min(t_{SSE}, t_{AVX}, t_{AVX2}) > 1.15$ where t_s is the execution time of setting s. Since the compiler flags for SSE, AVX, and AVX2 are identical to those for scalar, except for enabling various vector extensions, the performance difference is expected to be mainly from vectorization. Furthermore, a mutation's scalar performance may be significantly lower than that of the baseline. If so, the mutation might not be beneficial overall even if it has effective vectorization. Therefore, for this study, we are only interested in mutations that are both beneficial to the baseline while being vectorized effectively.

A. Effect of Mutations on Vectorization

Let's first investigate how mutations affect vectorization. The first row of Table VII lists the total number of loops that we studied for each compiler, denoted as L. The second row has the number and percentage of loops in L whose baseline are not effectively vectorized, denoted as N. It shows that ICC's vectorizer is the most effective among the three because it fails to vectorize the least percentage of L (59.6%). On the contrary, Clang's vectorizer is the least effective in the sense that it only manages to vectorize 21.1% of L effectively. The next row presents the number and percentage of loops in N that have beneficial mutations, denoted as B. Note that the percentages in this row (39.6%$\sim$47.5%) are much higher than the percentages of loops with beneficial mutation in L, which are 25.9%$\sim$36.6% (second row in Table III). This phenomenon indicates that loops that are not originally vectorized have higher chance to receive speedup from source-level transformations. Finally, the last row contains the number and percentage of loops in B whose beneficial mutations are effectively vectorized. It demonstrates that 36.1%$\sim$38.1% of the beneficial mutations are vectorized effectively while their baselines are not; thus, source-level transformations are able to increase performance by boosting compilers' vectorizers' success rate.

	GCC	ICC	Clang
# of loops studied (L)	1241	1175	1266
# (%) in L without effective vectorized baseline (N)	866 (69.8%)	700 (59.6%)	999 (78.9%)
# (%) in N that has beneficial mutation (B)	373 (43.1%)	277 (39.6%)	475 (47.5%)
# (%) in B whose beneficial mutation is effectively vectorized	141 (37.8%)	100 (36.1%)	181 (38.1%)

TABLE VII: Statistics of effective vectorization

B. Vectorization Settings

We compiled each mutation with various vectorization settings to assess compilers' effectiveness in (I) deciding whether to vectorize a vectorizable loop and (II) choosing the best vector extension for the task. Table VIII contains the number of loops (from the totals in the first row in Table VII) that are improved by at least 15% via changing vectorization settings. The first row focuses on the benefit by bypassing the profitability model. 8.1%∼9.4% of the loops can be sped up by over 15% with this method. The second row counts the loops whose performances rise by using an older vector extension. This time, 10.0%∼12.1% additional loops receive benefit. With the combination of the two efforts, 18.1%∼21.5% of the loops can receive a sizable performance boost without undergoing any transformations, as listed in the last row. We noticed that the numbers in the table from Clang are lower than the other two compilers', we believe the fact that Clang does not allow turning off profitability analysis explicitly contributes to this result.

	GCC	ICC	Clang
# (%) in L improved by bypassing profitability model only	117 (9.4%)	107 (9.1%)	102 (8.1%)
# (%) in L improved by selecting older vector extensions only	150 (12.1%)	138 (11.7%)	127 (10.0%)
# (%) in L improved by combining the two settings above	267 (21.5%)	245 (20.9%)	229 (18.1%)

TABLE VIII: Statistics of loops having speedup by changing vectorization settings

Figure 7 plots the speedup distribution of loops that are sped up by only changing the vectorization setting during compilation. While most speedups are below 2x, a number of loops gain speedups of 3x or above. Hence, a more accurate vectorization profitability model and a better understanding on the characteristics of different vector extensions can potentially help compilers to generate much faster results.

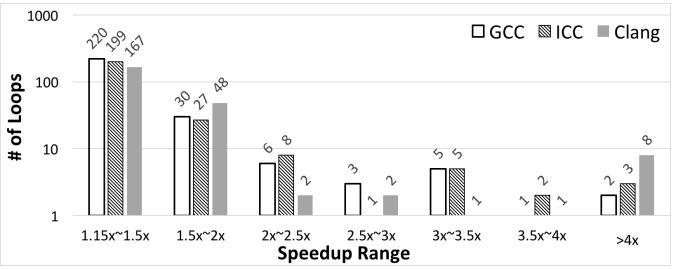


Fig. 7: Distribution of loops that are improved by changing vectorization settings when compiling the original loop

By combining the efforts of applying sequences of source-to-source transformations and searching for the best vectorization setting, we are able to accelerate 579 (46.7%), 420 (35.7%), and 589 (46.5%) loops for GCC, ICC, and Clang respectively, and the average speedup of these beneficial cases is 1.61x∼1.65x depending on the compiler.

C. Vectorization Profitability Case Study

In order to gain insight into the complex factors that affect the profitability of vectorization, we study the case in Listing 4, which is taken from NPB LU benchmark and whose scalar version outperforms its vectorized counterparts. The codelet's vectorized reference compilation is 2.2x slower than the scalar version where auto-vectorization is disabled on ICC. To understand the cause, we investigated the assembly code of both the reference and the scalar version. We discovered that the anomaly may be attributed to the following two reasons.

```
double ce[5][13], rsd[64][65][65][5];

for(i = 0; i < nx; i++) {
  iglob = i;
  xi = iglob /(nx0 - 1);
  for(j = 0; j < ny; j++) {
    jglob = j;
    eta = jglob /(ny0 - 1);
    for(k = 0; k < nz; k++) {
      zeta = k /(nz - 1);
      for(m = 0; m < 5; m++) {
        rsd[i][j][k][m] = ce[m][0] + ce[m][1] * xi
            + ce[m][2] * eta + ce[m][3] * zeta + ce
            [m][4] * xi2 + ce[m][5] * eta2 + ce[m
            ][6] *   zeta2 + ce[m][7] * xi3 + ce[m
            ][8] * eta3 + ce[m][9] * zeta3 + ce[m
            ][10] * xi4 + ce[m][11] * eta4 + ce[m
            ][12] * zeta4;
}}}}
```

Listing 4: NPB LU code

First, the reference code is vectorized at a length of 2. Instead of packing consecutive elements in the array `ce` to a SIMD register, the compiler unrolls the loop by a factor of 2 before vectorizing and then picks two adjacent elements in a column, e.g. `ce[0][0]` and `ce[1][0]`. Since one vector instruction can process operations from multiple iterations in the source code, we expected the number of assembly iterations in this vector code to be much less than that in the scalar one. However, we were astonished to see that the length of the scalar code is instead half of that of the vectorized code. By scrutinizing the assembly, we learned that the compiler fully unrolls the scalar code's innermost loop. It turns into fewer iterations and improves performance by eliminating the end-of-loop test. Meanwhile, the compiler aggressively schedules instructions for the scalar code after unrolling as there is no data dependence. This might be able to enhance instruction pipelines with the help of better ILP.

Second, this codelet tends to have many write cache misses since the `rsd` array does not fit into L1 and even L2 cache. Vector code is usually supposed to stress memory more than scalar code since it is more likely to complete computations faster. But it turns out that scalar code surprisingly manages to keep the memory much busier in this case. For example,

we found that the scalar code is able to fetch two cache lines concurrently for `rsd` over 14% of the execution while the vector code is essentially fetching one cache line at a time. Moreover, the scalar code keeps fetching at least one cache line over 70% of the execution while the vector code keeps the memory busy for only 31% of the execution. Since the most expensive factor is write cache misses, and the scalar version manages to process that more aggressively, it runs faster than the vector code. We observe that the vector code has more L1 hitting loads from `ce` in between write missing stores to `rsd`. These loads fill up the load buffer, cause the processor to stalls and prevent the processor from executing the stores more aggressively.

Consequently, compilers may not be able to accurately predict the outcome of vectorization due to complex factors interfering with each other.

VII. Related Work

The work that is most related to this study is the work of Maleki et al [28] who studied the effectiveness of vectorization in compilers as well as how transformations aid vectorization. Their study demonstrated, for vectorization, some of the instability effects discussed in our paper although they applied transformations by hand. There have been numerous studies on the selection of the transformations and the complexity of the space of performance obtained. Particularly influential is the work of Knijnenburg and O'Boyle [31], [32]. However, their work is more about the design of novel compiler algorithms than on how to evaluate compilers. Also related is the work on the selection of compiler options mentioned in the introduction [1], [2]. Aimed at accelerating performance evaluation of programs, a few prior works also proposed to extract the hotspots from applications [33], [34] and save them as stand-alone codelets. Castro et al [33] isolated code at the Intermediate Representation (IR) level using LLVM framework. In contrast, our extractor is implemented as a separate component of the *ROSE* compiler to isolate loop nests at the source level. Liao et al [34] also employed *ROSE* to develop their extractor. While they mainly focused on outlining the kernels of the target application at the function level for automatic kernel tuning and specialization, our extractor concentrates on isolating `for` loops. In addition, the goal of our extractor is to provide stand-alone codelets for loop transformation.

VIII. Conclusion

This paper aimed to investigate (I) potential room for improving the loop optimization passes of state-of-the-art compilers, (II) the stability of the compilers when dealing with semantically equivalent versions of a loop nest, (III) how source-level loop transformations affect the effectiveness of optimizations applied by today's compilers, (IV) the accuracy of compilers' vectorization profitability models.

We accomplished the goals by profiling an extensive collection of C `for` loop nests extracted from various sources, including benchmarks, libraries, etc., along with numerous mutations derived from them via semantic-preserving transformations applied by a *mutator*. From the profiling results, we analyzed the major effects of the transformations on performance by correlating the speedup with the change in performance metrics. Using the correlation and by manually inspecting the assembly code of interesting cases, we found that as the number and complexity of the optimization passes in modern compilers increase, the effect of source-level transformations becomes quite difficult to predict.

For example, a transformation may impact the same performance metric in opposite directions. One instance is unrolling, which may either help or hinder vectorization depending on the vectorization technique a compiler applies. To make the matter worse, when a compiler implements both loop and SLP vectorization (and potentially only applies one of them to a loop), the effect of unrolling on vectorization becomes unpredictable. Therefore, it is now harder for programmers (and source-level optimizer such as polyhedral compilers) to control the behavior of their code by altering the loop structure. Moreover, because different compilers may react to the same transformation in different ways, it is even harder for a programmer to write a loop structure that can be optimized well by multiple compilers.

During experiments, we noticed that, when the vectorization profitability model fails, the performance of a loop can be severely influenced. Also, the newest vector extension, although having longer vector length and more features than the older ones, can be outperformed by the older ones. Our results showed that by empirically choosing whether to vectorize a loop and/or the best vector extension for the said loop, 18.1%$\sim$21.5% of the loop nests can be sped up by at least 15% without modifying the source code.

We then quantified compiler stability by introducing a *stability score* and learned that the evaluated compilers are far from being stable. We also investigated if source-level transformations are able to narrow the performance gap among compilers and quantitatively confirmed it by devising a *convergence score*.

Finally, with the combined effort of applying source-level transformations and tuning vectorization settings, 35.7$\sim$46.5% of the loops are improved by over 15%, and a loop nest can expect a 1.61x$\sim$1.65x speedup on average if we manage to find a beneficial mutation and/or better vectorization setting for it. Because the results are only the lower bound of potential performance improvement, they prove that there is a significant performance headroom for each of the three compilers evaluated.

References

[1] Z. Pan and R. Eigenmann, "Fast, automatic, procedure-level performance tuning," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006, pp. 173–181.

[2] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO))*, 2003, pp. 204–215.

[3] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors," in *Fourth International Computer Software and Applications Conference*. IEEE, 1980, pp. 201–218.

[4] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle, *Iterative Compilation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 171–187. [Online]. Available: https://doi.org/10.1007/3-540-45874-3_10

[5] R. Allen and K. Kennedy, "Optimizing compilers for modern architectures a dependence-based approach," 2001.

[6] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[7] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, Mar. 1985.

[8] L.-N. Pouchet, "Polyopt/C: A polyhedral optimizer for the rose compiler," http://web.cse.ohio-state.edu/~pouchet/software/polyopt, 2011.

[9] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, "The ALPBench benchmark suite for complex multimedia applications," in *IEEE International Proceedings of the IEEE Workload Characterization Symposium (IISWC)*, 2005, pp. 34–45.

[10] "Asc benchmarks," https://asc.llnl.gov/sequoia/benchmarks/, 2008.

[11] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "Cortexsuite: A synthetic brain benchmark suite," in *IEEE International Proceedings of the IEEE Workload Characterization Symposium (IISWC)*, 2014, pp. 76–79.

[12] P. Rundberg and F. Warg, "The FreeBench v1. 0 benchmark suite," *URL: http://www. freebench. org*, 2002.

[13] R. F. Van der Wijngaart and T. G. Mattson, "The parallel research kernels." in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[14] "Livermoore loops," http://www.roylongbottom.org.uk/livermore\%20loops\%20results.htm, 1986.

[15] "Media bench ii," http://mathstat.slu.edu/~fritts/mediabench, 2006.

[16] S. Browne, J. Dongarra, E. Grosse, and T. Rowan, "The Netlib mathematical software repository," *D-Lib Magazine, Sep*, 1995.

[17] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The NAS parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[18] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[19] "SPEC benchmarks," http://www.spec.org, 2017.

[20] "Extended test suite for vectorizing compilers," http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz, 2011.

[21] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[22] "Xiph.Org Foundation," https://www.xiph.org, 2017.

[23] "The Lame Project," lame.sourceforge.net, 2017.

[24] "TwoLAME," www.twolame.org, 2017.

[25] "GAP," www.gap-system.org, 2017.

[26] "mozjpeg," github.com/mozilla/mozjpeg, 2017.

[27] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[28] S. Maleki, Y. Gao, M. J. Garzar, T. Wong, D. A. Padua *et al.*, "An evaluation of vectorizing compilers," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 372–382.

[29] S. Larsen and S. Amarasinghe, *Exploiting superword level parallelism with multimedia instruction sets*. ACM, 2000, vol. 35, no. 5.

[30] D. Nuzman and A. Zaks, "Outer-loop vectorization-revisited for short simd architectures," in *Parallel Architectures and Compilation Techniques (PACT), 2008 International Conference on*. IEEE, 2008, pp. 2–11.

[31] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle, "Embedded processor design challenges," E. F. Deprettere, J. Teich, and S. Vassiliadis, Eds. New York, NY, USA: Springer-Verlag New York, Inc., 2002, ch. Iterative Compilation, pp. 171–187.

[32] H. Leather, E. Bonilla, and M. O'Boyle, "Automatic feature generation for machine learning based optimizing compilation," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization(CGO)*, 2009, pp. 81–91.

[33] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby, "CERE: LLVM-based codelet extractor and replayer for piecewise benchmarking and optimization," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 1, pp. 6:1–6:24, 2015.

[34] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas, "Effective source-to-source outlining to support whole program empirical optimization," in *Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing (LCPC)*, 2010, pp. 308–322.