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

ZHANGXIAOWEN GONG, University of Illinois at Urbana-Champaign, USA
ZHI CHEN, University of California, Irvine, USA
JUSTIN SZADAY, University of Illinois at Urbana-Champaign, USA
DAVID WONG, Intel, USA
ZEHRA SURA, IBM, USA
NEFTALI WATKINSON, University of California, Irvine, USA
SAEED MALEKI, Microsoft, USA
DAVID PADUA, University of Illinois at Urbana-Champaign, USA
ALEXANDER VEIDENBAUM, University of California, Irvine, USA
ALEXANDRU NICOLAU, University of California, Irvine, USA
JOSEP TORRELLAS, University of Illinois at Urbana-Champaign, USA

Modern compiler optimization is a complex process that offers no guarantees to deliver the fastest, most efficient target code. For this reason, compilers struggle to produce a stable performance from versions of code that carry out the same computation and only differ in the order of operations. This instability makes compilers much less effective program optimization tools and often forces programmers to carry out a brute force search when tuning for performance. In this paper, we analyze the stability of the compilation process and the performance headroom of three widely used general purpose compilers: GCC, ICC, and Clang. For the study, we extracted over 1,000 `for` loop nests from well-known benchmarks, libraries, and real applications; then, we applied sequences of source-level loop transformations to these loop nests to create numerous semantically equivalent *mutations*; finally, we analyzed the impact of transformations on code quality in terms of locality, dynamic instruction count, and vectorization. Our results show that, by applying source-to-source transformations and searching for the best vectorization setting, the percentage of loops sped up by at least 1.15x is 46.7% for GCC, 35.7% for ICC, and 46.5% for Clang, and on average the potential for performance improvement is estimated to be at least 23.7% for GCC, 18.1% for ICC, and 26.4% for Clang. Our stability analysis shows that, under our experimental setup, the average coefficient of variation of the execution time across all mutations is 18.2% for GCC, 19.5% for ICC, and 16.9% for Clang, and the highest coefficient of variation for a single loop nest reaches 118.9% for GCC, 124.3% for ICC, and 110.5% for Clang. We conclude that the evaluated compilers need further improvements to claim they have stable behavior.

CCS Concepts: • **Software and its engineering → Software performance**; *Source code generation*;

Additional Key Words and Phrases: compiler stability, source-to-source transformation, vectorization

Authors' addresses: Zhangxiaowen Gong, University of Illinois at Urbana-Champaign, USA, gong15@illinois.edu; Zhi Chen, University of California, Irvine, USA, zhic2@ics.uci.edu; Justin Szaday, University of Illinois at Urbana-Champaign, USA, szaday2@illinois.edu; David Wong, Intel, USA, david.c.wong@intel.com; Zehra Sura, IBM, USA, zsura@us.ibm.com; Neftali Watkinson, University of California, Irvine, USA, watkinso@uci.edu; Saeed Maleki, Microsoft, USA, saemal@microsoft.com; David Padua, University of Illinois at Urbana-Champaign, USA, padua@illinois.edu; Alexander Veidenbaum, University of California, Irvine, USA, alexv@ics.uci.edu; Alexandru Nicolau, University of California, Irvine, USA, nicolau@ics.uci.edu; Josep Torrellas, University of Illinois at Urbana-Champaign, USA, torrella@illinois.edu.

2475-1421/2018/11-ART126
https://doi.org/10.1145/3276496

ACM Reference Format:
Zhangxiaowen Gong, Zhi Chen, Justin Szaday, David Wong, Zehra Sura, Neftali Watkinson, Saeed Maleki, David Padua, Alexander Veidenbaum, Alexandru Nicolau, and Josep Torrellas . 2018. An Empirical Study of the Effect of Source-Level Loop Transformations on Compiler Stability. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 126 (November 2018), 29 pages. https://doi.org/10.1145/3276496

1 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 today's compiler analysis and transformation technology rest on solid ground and are 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. This is why the standard way of selecting the best compiler command options is to use empirical methods (e.g. by Pan and Eigenmann [2006]; Triantafyllis et al. [2003]). In addition, because of the lack of understanding of the optimization process, there is much room for improvement even under the best compiler command settings as testified by the numerous projects that apply source-to-source pre-passes to improve the quality of the target code, such as the work by Kuck et al. [1980], Tiwari et al. [2011], Fursin et al. [2002], and Pouchet et al. [2008].

The benefits derived from tuning compiler options and applying source-to-source transformations are the results of sub-optimal compilers. A hypothetically "perfect" compiler would incorporate the best settings and transformation sequences in its optimization passes and would not need switch selection nor pre-passes. Such a "perfect" compiler would be "stable" in the sense that it would generate the same optimal target code for all semantically-equivalent versions of a loop nest. With near-optimal, stable compilers, programmers could focus on algorithm selection and program readability instead of having to twist the code with what often are obfuscating transformations to improve performance. The undecidability of program equivalence makes it impossible to develop compiler algorithms that generate the same code for all semantically-equivalent code sequences. However, there is no reason why they cannot generate target code that performs quite close to the optimum within a comfortable stability margin.

Although the compiler community is aware of the existence of instability, its magnitude has never been measured. In this paper, we present the first quantitative study on compiler stability. Because instability implies that there is often a performance headroom of the target code, we also study this headroom and estimate the performance improvement that could result from manipulating the source code. Since the majority of the work is usually performed in loops for compute-intensive applications, we choose to investigate the stability and performance headroom of the loop optimization passes of three popular compilers: *GNU Compiler Collection (GCC)*, *Intel C++ Compiler (ICC)*, and *LLVM C Compiler (Clang)*. The focus is on the compilation of `for` loops because it is among the language constructs whose analyses and transformations are best understood.

In order to make our empirical study as representative as possible, we built an extensive collection of loop nests extracted from 13 benchmarks suites and other sources, such as software libraries and machine learning kernels. The source code of these loop nests as well as their performance results are available in the *LORE repository* [Chen et al. 2017] developed to serve as a resource for the evaluation of compilers. We implemented an *extractor* to separate `for` loop nests from the original applications and build standalone *codelets* that executes independently. The codelets contain operations that measure execution time and read a number of performance counters. Only loop nests consuming more than 1,000 processor cycles were included in the quantitative study of stability and performance headroom. As a result, out of 3,197 loops that we have extracted, between 1,175 and 1,266 loop nests were investigated, depending on the compiler.

Our methodology for estimating compiler stability and performance headroom is to apply source-to-source transformations to obtain numerous semantically-equivalent versions of each loop nest, called *mutations* in this paper, and measure the variation in their execution time. Unlike the work mentioned above, our goal is not to improve the quality of the target code, but to study the effectiveness of today's compilers; therefore, we traverse the transformation space to create various loop structures without any performance target.

To generate the mutations, we developed an automated *mutator* that applies source-to-source transformations to the loop nests. These transformation sequences are combinations of five basic yet highly effective loop transformations: interchange, tiling, unrolling, unroll-and-jam, and distribution [Allen and Kennedy 2001]. Before applying any transformation, the mutator computes the dependences and determines whether or not the transformation can be applied; hence, any transformation sequence applied are semantic-preserving. From the loop nests that we studied, a total of 64,928~66,392 mutations, depending on the targeted compiler, were generated. The mutations were compiled by the three evaluated compilers. Because vectorization support is ubiquitous in modern processors and can have a significant impact on performance, the quality of a compiler's auto-vectorizer plays a significant role in the compiler's stability. Therefore, we also assessed the compilers' vectorization process by experimenting with different vectorization settings.

We quantified the stability of each compiler with an *intra-compiler stability score*. We found that the evaluated compilers are far from being stable and hence far from optimal. We also devised an *inter-compiler stability score* to measure the stability across multiple compilers, and we used it to confirm that source-level transformations, by moving the performance closer to the optimum, narrow the performance gap between compilers.

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 the room for improvement. Our results show that even though these transformations are likely implemented by the investigated compilers, their availability is not enough, and that the decision on whether or not to apply the transformation and the selection of the right parameters can significantly impact the performance of the resulting code. Although we can only obtain a lower bound of the performance headroom by trying limited combination of transformations, we expect this result, together with a figure of merit for stability, to be a useful indication of the distance from optimality. And, by repeating the measurements along the years, these values could give us a measure of progress.

Our results indicate 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 also analyzed how each of the five individual transformations applied by the mutator affects performance in order to find the deficiencies in the compilers that cause the instability. 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. To attenuate the cost of accessing the performance counters though system calls, we chose to consider only loops with an execution time longer than 10,000 cycles, reducing the number of loops considered in this part of the study to 768~817 depending on the compiler. We discuss the effect of transformations on specific loops to illustrate the complex ways in which they affect compiler output and the magnitude of the challenge faced by compiler writers in developing stable optimization strategies. For two of the transformations, we also propose ideas that may help compilers increase stability against them. The capability 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.

The rest of the paper is organized as follows. In Section 2, we describe how we extract loop nests and generate mutations from them. Sections 3 and 4 present the experimental settings and quantitative results, respectively. In Section 5, we analyze how different transformations affect performance. In Section 6, we further explore how vectorization settings impact performance. Finally, Section 7 discusses related work, and Section 8 presents our conclusions.

2 LOOP EXTRACTION AND MUTATION

To study compiler stability and performance headroom, we apply a variety of source-to-source transformations to each `for` loop nest from an extensive collection. To carry out the transformations, the `for` loop in each of these nests is required to be able to transform into the canonical form

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

We refer to such canonical form as "`for` loop" or simply "loop" in the rest of the paper unless specified otherwise. To build the collection of loops, an *extractor* outlines all qualified `for` loop nests from a variety of `C` language benchmark suites and libraries. Then, each loop nest is transformed by a *mutator* to generate various mutations. Finally, the execution time of each loop nest and of its mutations is measured, and the variation across mutations of each loop nest is computed.

We developed both the *extractor* and the *mutator* based on the *ROSE* source-to-source compiler infrastructure by Quinlan [2000]. This section presents a short description of these two components. For more information, the reader is referred to Chen et al. [2017].

2.1 The Extractor

The extractor encapsulates each `for` loop nest from an *original program* into a separate standalone program called a *codelet*. The extractor starts with finding all `for` loops in the original program by scanning the abstract syntax tree (AST). A `for` loop is skipped if it contains function call(s) other than standard math functions. If multiple `for` loops resemble a loop nest, the extractor identifies the outermost loop and generates a codelet for the entire loop nest only. Other types of loop such as `while` loops can be included as parts of a `for` loop's body, but our system does not processes them as the outermost loops of loop nests nor apply any transformation to them.

We choose to feed the codelets with the same data used in the original program to make both programs behave similarly as much as possible. To achieve that, the extractor instruments each loop nest to save the values of all read-only or write-after-read (WAR) variables in the loop right before executing the loop. Input data from global/static variables, heap, and stack are handled separately and will be restored to their corresponding locations when later executing the codelet. The loop bounds are recorded at run-time if their values are not constants. Finally, the instrumented original program is executed to create an input data file for each loop nest.

If a loop nest has multiple execution instances (e.g. inside a function that is invoked multiple times), the extractor saves data from one of the executions chosen using the reservoir sampling algorithm, which grants each execution equal chance to be selected [Vitter 1985].

To create the codelet, the extractor copies the source code of the loop nest from the original program and surrounds it with the following collection of operations:

(1) Read the input data file and initialize variables and memory regions with the values they had during the execution of the loop nest in the original program.
(2) Record time using the `RDTSCP` instruction, which allows accurate timing measurements with a resolution of just two instructions as reported by Paoloni [2010].
(3) Read hardware performance counter values.

(4) Use all the data that the loop nest produces to generate reduced values that are output to I/O so that the compilers do not remove operations as dead code. For a scalar variable, the codelet writes the variable value; for an integer array, the codelet writes the MD5 hash of the array; and for a floating point array, the codelet writes its sum reduction.
(5) Repeatedly execute the loop nest 100 times and record the median of the execution time. Variables and memory regions are reinitialized before each re-execution.

The codelets are thus completely self-contained and ready for transformation by the mutator, compilation, and execution.

Although the source code and input for an extracted loop are replicated from the original program, the loop's behavior during the codelet execution may vary from its behavior during the original program execution because (I) the cache state will typically differ from the state during the execution of the original program. On the other hand, it will be consistent across re-executions of the loop in the codelet (except for the first execution); (II) some of the compiler's inter-procedural and/or inter-loop analysis may lead to changes in the optimization process. For example, the extractor outlines neighboring outermost loops into separate codelets and therefore disables any interaction between them, but a compiler may fuse them when compiling the original program. However, these differences are acceptable since our study focuses on how performance varies between the original codelet and its mutations. Thus, replicating the exact cache state and the optimizations applied on the original program is not essential because our goal is not to optimize the original program.

2.2 The Mutator

The extracted loop nests in the form of codelets are processed using the mutator to create semantically equivalent mutations. The mutator applies sequences of source-to-source loop transformations that are constructed from interchange, tiling, unrolling, unroll-and-jam, and distribution. These relatively easy to implement transformations are among the best-known loop transformations, so they can be effortlessly added to any compiler if they are not currently present.

We imposed limitations on the transformation sequences because the number of mutations of a single loop nest may grow exponentially with the number of transformations and the number of possible parameters to each transformation [Tiwari et al. 2011]. These limitations ensured the number of mutations generated remained reasonable.

First, the mutator does not explore the transformation space exhaustively. Instead, it applies sub-sequences of transformations of the following sequences:

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

A sub-sequence can skip transformation(s) in the above sequences. For example, $interchange \rightarrow distribution$ is a valid sub-sequence. However, the order of transformation needs to be preserved. For instance, $distribution \rightarrow interchange$ is never applied. We chose this ordering to ensure that transformations that only operate on perfectly nested loops (i.e. all assignment statements are in the innermost loop) due to the limitation of our tool, namely interchange, tiling, and unroll-and-jam are not applied after any transformation that may render loop nests imperfect, namely distribution, unrolling, and unroll-and-jam. The maximum length of transformation sequence is 4.

Second, we limit the parameters to each transformation as shown in Table 1. 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 the parameters are the size 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 the same number of times. 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 dependence information; hence, distribution does not take any parameter. As presented in column 2, we only use selected tile sizes/unroll factors for tiling, unrolling, and unroll-and-jam. If the loop level being transformed has a static trip count that is smaller than one of the tile sizes/unroll factors, the transformation with the corresponding parameter is skipped. For example, the mutator does not apply unroll by 8 times on the loop `for(i=0;i<=5;i+=1){...}`, which has a static trip count of 6.

Table 1. Transformations and their parameters

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

Although these limitations confine 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 at finding lower bounds for performance headroom and instability of the investigated compilers.

In addition to the imposed restrictions, the number of mutations is also limited by data dependence. Unrolling is the only employed transformation that is guaranteed to be semantics-preserving (although additional care needs to be taken when `continue`, `break`, and/or `goto` is present in the loop body). To ensure the legality of the other four transformations, the mutator analyzes dependence through *PolyOptC* by Pouchet [2011], which provides an interface between *ROSE* and the polyhedral model based dependence analyzer *Candl* by Bastoul and Pouchet [2012]. If a loop nest is incompatible with the polyhedral model, e.g. because of having non-affine array subscripts, the mutator applies only unrolling to it because *Candl* cannot analyze it. Therefore, all generated mutations are guaranteed to preserve the original semantics in theory. In practice, to counteract any potential bug in the mutator or the compilers, we also added a second layer sanity check. As described in Section 2.1, a codelet writes the reduction of all of the loop's output to I/O. We compare the original loop nest and its mutations' outputs to verify semantic equivalence. Although this method may produce false positives or false negatives, we found it sufficient during our experiments.

The nesting depth and the dependence graph of a loop nest determine how many mutations are produced from it. Among the loops that we studied, up to 1,680 mutations and on average 60 mutations were created from a single loop nest.

2.3 Collection of Loop Nests and Their Mutations

We extracted 3,197 loop nests from various sources for this study, such as: benchmarks, audio/video codecs, and machine learning kernels. The first column in Table 2 lists the sources that we extracted the loop nests from. For the benchmarks, we used the default dataset during extraction except for SPEC and NPB. We used the "ref" dataset for SPEC and the "CLASS=B" dataset for NPB. For the libraries, we used the test data that they provided. In total, we produced 100,219 mutations from the 3,197 codelets; however, we only study the results from loops whose execution time exceeded 1,000 cycles. The final number of loops and mutations that we studied is presented in Table 2 and will be discussed in Section 4.

Table 2. The numbers of loop nests and their mutations included in the study

Benchmark	# of loops (# of mutations)		
	GCC	ICC	Clang
ALPBench [Li et al. 2005]	24 (72)	22 (66)	31 (129)
ASC Sequoia [LLNL 2008]	22 (350)	21 (347)	22 (350)
Cortexsuite [Thomas et al. 2014]	60 (1060)	57 (791)	62 (1042)
FreeBench [Rundberg and Warg 2002]	38 (242)	31 (141)	39 (245)
Parallel Research Kernels (PRK) [Van der Wijngaart and Mattson 2014]	36 (286)	23 (189)	34 (261)
Livermore Loops [Peters 1992]	53 (1443)	51 (1436)	57 (1612)
MediaBench II [Fritts et al. 2009]	152 (773)	120 (532)	183 (1279)
Netlib [Browne et al. 1995]	25 (207)	21 (195)	24 (204)
NAS Parallel Bench. (NPB) [Bailey et al. 1991]	196 (52259)	195 (52244)	198 (52350)
Polybench [Pouchet 2012]	90 (3574)	91 (3589)	91 (3589)
SPEC 2000 [Henning 2000]	122 (1263)	125 (1272)	129 (1337)
SPEC 2006 [Henning 2006]	102 (421)	103 (425)	129 (907)
Extended TSVC [Maleki et al. 2011]	149 (1955)	149 (1955)	149 (1943)
Machine learning kernels [Redmon 2016]	27 (177)	27 (177)	21 (123)
Libraries [GAP 2007; LAME 2017; Mozilla 2017; TwoLAME 2017; xiph.org 2017]	145 (1735)	139 (1569)	97 (1023)
Subtotal	1241 (65817)	1175 (64928)	1266 (66392)
Execute for over 1,000 cycles for all 3 compilers	1061 (63902)		

Different benchmark applications may have similar loops (e.g. matrix multiplication kernel). We did not attempt to group similar loops and pick one to represent the group, so the results may bias towards larger clusters of similar loops. However, we believe it is acceptable because it applies a natural weight on the loop types that are more common and thus need more attention.

3 EXPERIMENTAL SETUP

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 *LLVM C Compiler (Clang)* 4.0.0.

The experiments were conducted on an Intel Xeon E5-1630 v3 processor (Haswell microarchitecture, 32KB/32KB private L1 data/instruction cache, 256KB private L2 cache, 10MB shared L3 cache) with 32GB DDR4 2133 RAM. The CPU is equipped with an invariant time stamp counter (TSC) so that the readout from `RDTSCP` is accurate regardless of ACPI P-, C-, and T-states according to Intel [2016]. To achieve stable results, all executions were assigned to the same core with dynamic frequency scaling, Intel *Hyper-Threading*, C-State higher then one, and *TurboBoost* technologies disabled. The experimental results from a single machine setup are sufficient to provide an estimation of compiler stability and performance headroom since we believe similar traits may exist on other systems. However, doing experiments on a single hardware setup tends to add measurement bias towards certain compiler(s) [Mytkowicz et al. 2009]. As a result, the difference in the tested compilers' stability and performance headroom reported by our results should not be considered as a conclusive comparison of the compilers' quality because such difference may vary on other systems. In the future, conducting further experiments on additional machine setups may reduce the measurement bias, providing more comprehensive conclusions.

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 inter-procedural 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.

4 RESULTS

This section presents the main results of our study. In Section 4.1, we report the performance of the code generated by the evaluated compilers from the original loop nests. We discuss the overall effect of the source-to-source transformations applied by the mutator on performance in Section 4.2, discuss the performance impact from the length of transformation sequence in Section 4.3, evaluate the effect of each transformation in Section 4.4, discuss the performance headroom of the loops from different benchmarks in Section 4.5, and finally propose metrics to measure compiler stability and performance convergence in Section 4.6 and 4.7, respectively.

4.1 Baseline Performance

Table 2 lists, for each of the three compilers considered, the number of loops and and the number of mutations used for the part of the study presented in this section. Recall that we require the execution time of a loop nest to be at least 1,000 cycles; therefore, only between 1,175 and 1,266 loop nests are used for evaluating each of the three compilers.

To compare the effect of the compilers on the baseline loops, we only consider the 1,061 loops whose execution time is longer than 1,000 cycles for all three compilers. On average, the code generated by GCC and Clang is 1.06x and 1.27x slower respectively than that by ICC; however, they generate code that outperforms ICC's by at least 15% in 174 and 114 cases, respectively. Therefore, the optimal compiler for each loop varies. We chose a 15% threshold because it is a meaningful difference, even with experimental timing noise; this threshold was also used by Maleki et al. [2011].

4.2 Overall Impact of Our Collection of Mutations on Performance

We calculate the speedup of a loop nest l's fastest mutation over its baseline by

$$speedup^{(l)} = \frac{min(t^{(l)}_{mutation[0]}, ..., t^{(l)}_{mutation[n^{(l)}-1]})}{t^{(l)}_{baseline}} \quad (1)$$

In the formula, $t^{(l)}_{baseline}$ and $t^{(l)}_{mutation[i]}$ are the execution times of a compiler's outputs of baseline loop l and its mutation $i \in [0, n^{(l)} - 1]$ respectively, and $n^{(l)}$ is the number of mutations generated from baseline loop l, which varies depending on the loop. We see that, on average, the fastest mutation of a loop is 1.11x, 1.05x, and 1.16x faster than the baseline for GCC, ICC, and Clang respectively, as shown in Table 3 row 2. Also, the standard deviations of speedup are 1.02~1.04, suggesting that the range of speedup is significant.

Table 3. General statistics of mutations' performance impact

		GCC	ICC	Clang
1	# of loops studied (L)	1241	1175	1266
2	μ_g (σ_g) of the fastest mutation to baseline speedup	1.11 (1.02)	1.05 (1.04)	1.16 (1.03)
3	average lower bound of performance headroom	16.7%	13.8%	20.9%
4	# (%) in L that have beneficial mutation(s)	402 (32.4%)	304 (25.9%)	463 (36.6%)
5	# (%) in L that have all mutations unfavorable	89 (7.2%)	188 (16.0%)	73 (5.8%)

We report the average lower bound of performance headroom by applying source-to-source transformations in row 3, which is calculated by

$$headroom \geq ((\prod_{l=1}^{L} \frac{min(t^{(l)}_{baseline}, t^{(l)}_{mutation[0]}, ..., t^{(l)}_{mutation[n^{(l)}-1]})}{t^{(l)}_{baseline}})^{\frac{1}{L}} - 1) \times 100\% \quad (2)$$

When computing the lower bound of headroom, if all mutations are slower than the baseline, we consider it to be 0%. On average, the performance of a loop nest has a headroom of at least 13.8%~20.9%, depending on the compiler.

We assign categories to each mutation based on its 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*, the rest are considered to be *neutral*. As shown in row 4, 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, the percentage of loops that only have unfavorable mutations is much higher for ICC at 16.0% vs. Clang at 5.8% and GCC at 7.2%.

Focusing on loops with beneficial mutations, we get the distribution of speedups shown in Figure 1. The plot reveals that for all three compilers, although the majority of the speedups are below 2x, a number of loops receive an over 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* 8 times, Clang decides to further fully unroll the loop and pre-calculates most of the scalar operations at compile time, accelerating the loop by 20x. Nevertheless, there is a case where interchange facilitates better locality and vectorization to help a loop nest from *TSVC* gain 15x performance with Clang.

While ICC has fewer loops that are sped up by the source-to-source transformations, the number of loops that have an over 3x speedup is comparable to Clang's and is much greater than GCC's. Furthermore, both ICC and Clang on average obtain a 1.54x maximum speedup for the loops with beneficial mutation(s) whereas GCC on average can only attain a 1.46x maximum speedup for loops in that category.

Next, we consider loops where all mutations are unfavorable. Figure 2 shows the distribution of loops at various slowdown ranges. 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 slowdown factors are less than 2x, but for several loops, slowdowns are greater than 4x and can be up to 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.

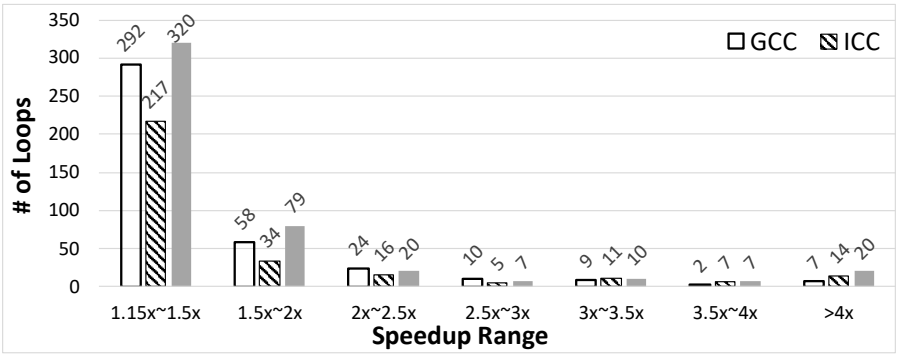


Fig. 1. Distribution of loops with beneficial mutation(s) and their speedup

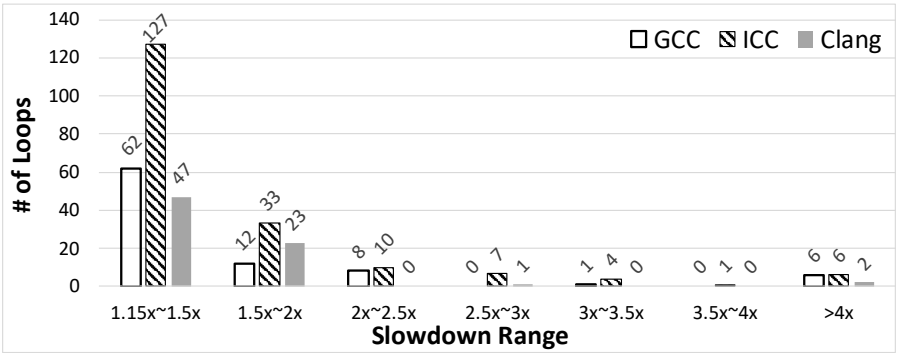


Fig. 2. Distribution of loops with all mutations unfavorable in different slowdown range

4.3 Performance Impact from Different Transformation Sequence Lengths

Table 4 shows the statistics of the performance impact from different transformation sequence lengths. If a given loop nest has mutation(s) that are transformed by s number of transformations, the performance of the fasted mutation among them is used in the calculation of the statistics for sequence length s.

Table 4. Statistics of the performance impact of the length of transformation sequence

Seq. length	μ_g of speedup			% of loops benefited (B)			μ_g of speedup in B		
	GCC	ICC	Clang	GCC	ICC	Clang	GCC	ICC	Clang
1	1.07	1.02	1.11	29.2%	22.0%	32.0%	1.45	1.55	1.51
2	0.99	0.95	1.06	26.5%	21.1%	35.1%	1.46	1.47	1.55
3	1.01	0.98	1.10	26.6%	22.8%	31.9%	1.54	1.52	**1.79**
4	**0.53**	**0.59**	**0.61**	**2.2%**	**0.0%**	**6.4%**	**1.18**	N/A	**2.06**

The first set of columns lists the average speedup per sequence length for each compiler. It reveals that for sequence length of 1~3, on average there is at least one mutation that performs on par with the baseline for each affected loop nest. However, for sequence length of 4, the fastest mutation is expected to only have 0.53x~0.61x of the baseline's performance depending on the

compiler, as highlighted in the table. This is because as the number of transformations applied increases, the structure of the loop nest becomes more complicated, therefore making the compilers hard to analyze and optimize. The second set of columns lists the percentage of loops affected by a given sequence length that have beneficial mutation(s) of that sequence length. When the sequence length is lower than 4, the percentages are relatively stable for all three compilers, but for sequence length of 4, as highlighted, none of the mutations are beneficial for ICC, and the percentages for GCC and Clang are also very low at 2.2% and 6.4% respectively. The last set of columns lists the average speedup that the loops with beneficial mutations of a given sequence length get from the fastest mutation of that sequence length. When the sequence length is lower than 4, the expected speedup is around 1.5x for all three compilers except when the sequence length is 3, Clang obtains a notably higher speedup of 1.79x. When the sequence length is 4, GCC on average only receives a 1.18x speedup, but surprisingly for Clang, the expected speedup is as high as 2.06x.

The results suggest that (I) by applying a single transformation to a loop nest, we can already expect comparable speedup with applying multiple transformations; (II) Clang may receive more benefit from longer transformation sequences than ICC and GCC do; (III) increasing the sequence length beyond 4 is unlikely to yield better results.

4.4 Performance Impact from Each Transformation

Table 5 shows the statistics of the performance impact from each loop transformation. If a transformation T is legal for a given loop nest, the performance of the mutation that is the result of applying only T with the parameters that produce the highest performance is used in the calculation of the statistics for T. Note that the fastest variant of a transformation may still be slower than the baseline. The table reports the geometric mean and standard deviation of speedup as well as the percentage of the loop nests that have at least a beneficial variant of the transformation. In general, all three compilers react to the same transformation similarly with some exceptions that are highlighted in the table, which are: (I) while GCC and Clang on average show a speedup from unrolling, ICC shows a slowdown; (II) distribution is able to help ICC much more than it can help the other two compilers; (III) unroll-and-jam and tiling can speed up significantly more loops for Clang than for ICC and GCC; (IV) compared with GCC and Clang, ICC has less loops that benefit from interchange, unrolling, and unroll-and-jam, but more loops that benefit from distribution.

Table 5. Statistics of speedup from different transformations

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

While unrolling, unroll-and-jam, and distribution increase performance on average, the other two transformations, interchange and tiling, produce a slowdown on average. The intuitive reason is that most loops are already written with good locality, so altering the loop shape may lead to sub-par results from unstable compilers. Nonetheless, 6.4%~9.0% of the interchanged mutations and 9.2%~16.2% of the tiled mutations were beneficial, depending on the compiler.

4.5 Performance Headroom of the Loops from Each Benchmark

Figure 3 shows that loops from different benchmarks have varied performance headroom from applying source-level transformations. Among the benchmarks, loops from *polybench* and the libraries have over 20% headroom with all three studied compilers. Source-to-source transformations can accelerate loops from *polybench* with ease because this benchmark is designed for polyhedral compilers, which often are also source-to-source, to optimize the loop nests inside them. Loops from the libraries, on the other hand, have higher headroom because their loop types may be different than those in well-known benchmarks that are intensively studied by the compiler developers.

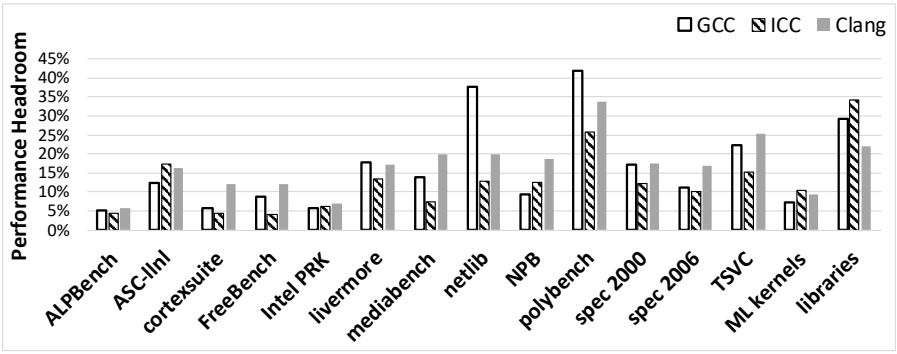


Fig. 3. Average performance headroom available from applying simple source-level transformations

Among the studied compilers, ICC has the lowest headroom while Clang has the highest headroom for most of the benchmarks. This is mainly because ICC is more aggressive in optimization compared with GCC and especially with Clang. It affects the results in two ways: (I) heavy optimization may leave less room for improvement; (II) ICC may sometimes undo source-level transformations. There are cases where it re-rolls or permutates the loop back after we apply unrolling or interchange on a loop. Nevertheless, there are notable exceptions: for the libraries, ICC's headroom (34%) is significantly higher than Clang's (22%); for *netlib*, GCC's headroom (38%) is much higher than ICC's (13%) and Clang's (20%); for *polybench*, GCC also has a headroom (42%) noticeably higher than ICC's (26%) and Clang's (34%).

4.6 Intra-compiler Stability

We expect a *perfect* compiler to undo unfavorable transformations and apply beneficial transformations to any mutation of a loop. We call such a compiler *stable* since it would produce the same performance for any mutation of a given loop. We do not expect a perfectly stable compiler to be built in the near future, and perhaps it will never be built. However, we hope that it will be possible to get very close to the perfect compiler, and the first step towards it is to quantitatively measure the stability of compilers. Therefore, we devised the following *intra-compiler stability score* S_{intra}:

$$S^{(l)}_{intra} = C_v(t^{(l)}) = \frac{\sigma(t^{(l)}_{baseline}, t^{(l)}_{mutation[0]}, \ldots, t^{(l)}_{mutation[n^{(l)}-1]})}{\mu(t^{(l)}_{baseline_l}, t^{(l)}_{mutation[0]}, \ldots, t^{(l)}_{mutation[n^{(l)}-1]})}$$
$$S_{intra} = \frac{1}{L}\sum_{l=1}^{L} S^{(l)}_{intra} \quad (3)$$

We compute $S^{(l)}_{intra}$, the intra-compiler stability of a loop l, as the coefficient of variation of the execution time of its n mutations and its baseline. The coefficient of variation is calculated by scaling the standard deviation of execution time with the average execution time; thus, it approaches 0 if a compiler produces perfectly stable performance for a given loop semantics. We further calculate the intra-compiler stability score of a set of L loops by taking the mean of $S^{(l)}_{intra}$ for all $l \in L$. The stability score has no absolute meaning by itself. Instead, it can be used to compare the stability of different compilers or to track the change in stability between different versions of a given compiler.

The intra-compiler stability score reflects a compiler's ability to recognize optimization opportunities. For example, a stable compiler would interchange back a loop that was interchanged by the mutator if this reversal improves memory access patterns and/or creates vectorization opportunities, a trait that we did occasionally observe from the studied compilers. Table 6 presents the S_{intra} and the highest $S^{(l)}_{intra}$ calculated from all transformation sequences as well as from individual transformations. Because all transformations are not valid for all loops, the table lists the number of loops included for calculating the stability score for each transformation. In each row, the compiler that produces the highest S_{intra} is highlighted. When considering all transformation sequences, the scores are 0.195, 0.182, and 0.169 for ICC, GCC, and Clang respectively. By definition, a higher stability score reflects greater instability; therefore, among the compilers we tested, ICC is the most unstable overall and Clang is the most stable, with GCC somewhere in between them.

Although the stability score suggests that Clang is more stable than the other two compilers, earlier in Section 4.2 we showed that source-to-source transformations are, on average, more beneficial to Clang than to ICC and to GCC. Intuitively, this may be because Clang is younger than the other two compilers, so it has a less aggressive yet also less brittle optimization process. This means that it is more stable for the limited set of transformations that we applied in general; however, it benefits more from better structured source code because of its relatively naïve optimization process. Consequently, the mutations compiled by Clang may have a lower performance variation but skew more towards speeding up the original loop.

Table 6. Stability scores for different transformations with the most unstable compiler being highlighted

Transformation	**# of loops included**	S_{intra} **(highest** $S^{(l)}_{intra}$**)**		
		GCC	**ICC**	**Clang**
All possible sequences	1061	0.182 (1.189)	**0.195 (1.243)**	0.169 (1.105)
Interchange	169	0.355 (0.866)	0.348 (0.879)	**0.385 (0.881)**
Tiling	430	**0.209 (1.036)**	0.208 (1.169)	0.190 (1.176)
Unrolling	1061	0.097 (1.175)	**0.123 (0.893)**	0.099 (0.788)
Unroll & jam	177	0.112 (0.511)	0.107 (0.524)	**0.137 (0.571)**
Distribution	106	0.111 (0.508)	**0.140 (0.681)**	0.099 (0.523)

More interesting observations are made from the stability scores of individual transformations. For interchange, the scores from all three compilers are much higher than their overall scores, indicating that interchange produces significant performance variations. Also, Clang is the least stable toward interchanged loop nests, and ICC is the most stable one against interchange. We believe it is due to ICC having a higher tendency to permute the loop nests, as mentioned in Section 4.5. For tiling, the stability scores for ICC and GCC are comparable, but the score for Clang is noticeably lower. For unrolling, unroll-and-jam, and distribution, the stability scores for all three compilers are significantly lower than their overall scores; thus, these three transformations cause less performance variations than the other two. Nevertheless, ICC appears to be less stable than

the other two compilers against unrolling and distribution, and Clang is less stable when dealing with unroll-and-jam. The per-transformation results suggest that different compilers may have strengths and weaknesses in terms of stability when facing different source-level transformations.

To demonstrate the instability, we plot in log scale the ratio of the execution time of each loop's fastest mutation and its slowest one (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-axis 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 when taking all transformation sequences into consideration.

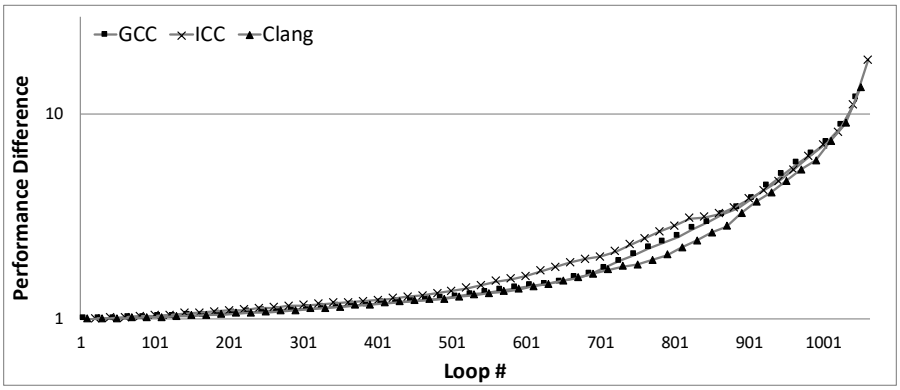


Fig. 4. Performance difference between each loop's best-to-worst mutations

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

4.7 Inter-compiler Stability

In Section 4.1, we mentioned that the performance of the code generated by the three compilers from the same loop may vary, meaning that apart from the intra-compiler stability that we discussed earlier, there is an additional inter-compiler stability to be concerned about. To quantify it, we propose an *inter-compiler stability score* S_{inter} to measure how close the effectiveness of the optimization processes of a set of compilers is, as given by,

$$S_{inter} = \frac{1}{L}\sum_{l=1}^{L}\frac{\sigma(t_C^{(l)})}{\mu(t_C^{(l)})} \quad C \in \{GCC, ICC, Clang\} \tag{4}$$

In the formula, $t_C^{(l)}$ is the execution time of baseline loop l compiled by compiler C. Like the S_{intra}, S_{inter} is an average of coefficient of variation. $S_{inter}^{(l)}$ of an individual loop l is computed by first taking the standard deviation of the execution time of the code generated by each compiler and then scaling it using their average execution time, so it measures how close the performances of a set of compilers' generated code are. S_{inter} of a set of loops L is the average of $S_{inter}^{(l)}$ where $l \in L$. The score is meaningful only when used in comparison, and a lower S_{inter} means closer performance. By comparing S_{inter} of different generations of the same set of compilers, one can track the progression of compiler stability in a bigger picture. However, in this paper we study only a single version of each compiler, so we use the score for another purpose: to investigate how

source-to-source transformations impact inter-compiler stability. We define the post-transformation inter-compiler stability score S'_{inter} as:

$$t_C'^{(l)} = min_C(t^{(l)}_{baseline}, t^{(l)}_{mutation[0]}, \ldots, t^{(l)}_{mutation[n^{(l)}-1]})$$
$$S'_{inter} = \frac{1}{L}\sum_{l=1}^{L}\frac{\sigma(t_C'^{(l)})}{\mu(t_C'^{(l)})} \quad C \in \{GCC, ICC, Clang\} \tag{5}$$

When calculating $S'^{(l)}_{inter}$ for a single loop, instead of using the execution time of the baseline loop, we use the execution time of the fastest mutation if it is faster than the baseline. Note that the fastest mutation of a given loop can vary when compiled by different compilers. For example, while interchange helps GCC to achieve the best performance for a given loop, unroll-and-jam can be the key performance enhancer for ICC for the same loop. Therefore, S'_{inter} calculates the inter-compiler stability as if the compilers have incorporated the proper transformation.

From our results, the baseline S_{inter} is 1.39, and the post-transformation S'_{inter} is 0.96. The lower S'_{inter} tells that by applying simple source-to-source transformations, the performance gap among the code generated by various compilers is narrowed; hence, source-level transformations can help the compilers that lag behind catch up. We call this phenomenon *the convergence effect.*

5 EFFECT OF TRANSFORMATIONS

To aid compiler developers in the design of more stable compilers, we studied the effects of source-level transformations on the studied compilers. In this section, we investigate how each of the five source-to-source transformations impacts performance by computing the correlation coefficient between performance change and each of a number of hardware performance counter readouts. If a strong correlation between performance and a metric is discovered for a transformation, we can derive the dominant effect produced by the transformation and therefore pinpoint the deficiency in the compilers that causes the instability. In addition, we also present a few case studies illustrating the complexity of the interaction between the transformations applied by the mutator and the compilers. We focused on the individual transformations because performing correlation analysis on all possible transformation sequences that the mutator generates is impractical.

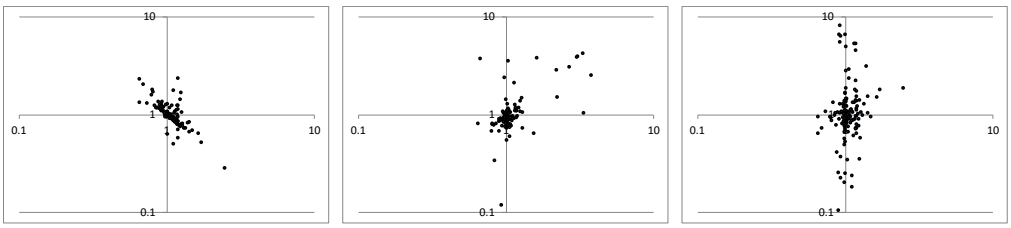
5.1 Computing Correlation Coefficients

We compute the correlation coefficient between (a) the change in value of a performance metric and (b) the change in execution time, where "change" refers to the difference between the original loop and the transformed loop. Each performance metric is obtained by reading a hardware performance counter or by computing from multiple 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, that is an application of T. Recall that transformations can produce multiple mutations since they are controlled by parameters (Table 1). (II) Compute the ratio of the execution time of 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 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 for loops with short execution time since the hardware counter reading process, which involves system calls, is also partially included in the measurement. Hence, to attenuate the inaccuracy, we only include loops with baseline execution time higher than 10,000 cycles. As a result, depending on the compiler, 768~817 loops are included

in this part of the study. Because the general performance statistics and the counter correlation analysis are decoupled, it is acceptable to use a subset of the loops for the latter.

Note that $\rho_{1/S,R} \in [-1, 1]$, and -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. On the other hand, if $|\rho_{1/S,R}|$ is insignificant for any of the performance metrics, either the transformation has multiple reasons that impact performance, or the reason is not captured by the limited set of performance metrics that we gather, so its effects are less clear.

Figure 5 illustrate these ideas. It plots the ratio of performance factor values R (y-axis) against speedup S (x-axis) in logarithmic scale. For a perfect correlation (i.e. $\rho_{1/S,R} \in \{-1, 1\}$), 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 (-0.79), and the points resemble a line $R = kS$ with $k < 0$ with a few noises. Figure 5 (b), on the other hand, presents a mid-range positive correlation $\rho_{1/S,R} = 0.27$. We can still see a $R = kS$ 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) $\rho_{1/S,R} = -0.79$ (b) $\rho_{1/S,R} = 0.27$ (c) $\rho_{1/S,R} = 0.03$

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 7, where derived metrics are highlighted.

5.2 Interchange

Loop interchange can convert non-unit-stride array access to unit-stride, which improves spatial locality. In addition, such a conversion may remove the need of gather-scatter instructions when vectorizing the loop.

The correlation values 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 cache behavior (`l1d.replacement`, `l1d_pend_miss.pending`, `l1d_pend_miss.pending_cycles`), and in L2 cache behavior (`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 (-0.7∼-0.4) with these cache related metrics. Moreover, interchange also affects TLB performance indicated by the -0.7∼-0.6 negative correlations with TLB related metrics (`dtlb_load_misses.miss_causes_a_walk`, `dtlb_load_misses.stlb_hit`). The locality improvements subsequently contribute to an increase in instruction execution rate suggested by moderate positive correlations (0.2∼0.5) with `inst_rate`.

Table 7. Top performance metrics that correlate to execution time

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 (ILP)
l2_rw_rate	L2 read/write rate calculated by dividing the total number of L2 requests by the cycle count

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 hand analyzing 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 for each of the three compilers. ICC gives the best performance for the original loop nest; Clang and GCC's outputs are respectively 1.13x and 1.45x slower than ICC. 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's executes 384K instructions. On the

other hand, ICC vectorizes the loop using gather-scatter and its output executes 288K instructions. However, the speedup from vectorization is modest due to the gather scatter operations.

After the mutator interchanges the loop nest as shown in Listing 1, the accesses to `Px` and `Vy` become unit-stride, and `Cx` becomes a loop constant of the innermost loop. As a result, all three compilers vectorize the mutation without gather-scatter. 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. Additionally, after interchange, Clang's mutation takes the lead and becomes 1.46x faster than ICC's output and 2.39x faster than GCC's.

We confirmed that the compilers perform loop interchange in some cases by scrutinizing the assembly. However, the high correlations with locality related metrics suggest that overall they rarely apply it, implying that these compilers fail to do interchange properly is due to inaccurate profitability models and/or not seeking interchange opportunities most of the time. In fact, some extracted loops have embarrassingly bad permutations that can be manually spotted instantly. For example, the loop nest in Listing 2 from *NPB* has an obviously problematic memory access pattern, and it does not contain loop carried dependence. However, none of the three compilers interchanges it to a better shape, and the properly interchanged mutation becomes 2.6x~3.2x faster depending on the compiler. Such scenario is common in real world when the programmer does not pay attention to performance, or when the source code is generated by tools such as f2c [Feldman 1990]. Hence, it would be helpful if compilers prioritize the analysis for interchange opportunities.

```
double dt, rhs[65][65][65][5];

for(j = 1; j <= y - 2; j += 1)
  for(k = 1; k <= z - 2; k += 1)
    for(m = 0; m < 5; m += 1)
      for(i = 1; i <= x - 2; i++)
        rhs[i][j][k][m] = rhs[i][j][k][m] * dt;

/* interchanged */
for(i = 1; i <= x - 2; i++)
  for(j = 1; j <= y - 2; j += 1)
    for(k = 1; k <= z - 2; k += 1)
      for(m = 0; m < 5; m += 1)
        rhs[i][j][k][m] = rhs[i][j][k][m] * dt;
```

Listing 2. Original and interchanged *NPB BT* code

5.3 Unrolling

Loop unrolling is a technique traditionally used to obtain better performance at the expense of program size. Unrolling may improve performance by reducing control overhead (e.g. advancing the iterator and testing exit conditions), exposing more instruction level parallelism (ILP), and/or by enabling scalar optimizations (e.g. common sub-expression elimination, constant folding, etc). Compilers may undo a source level unrolling by applying loop re-rolling.

The correlation results demonstrate that unrolling does reduce dynamic instruction counts. All three compilers show moderate to high negative correlations (-0.7~-0.3) with `inst_retired.any`. We also see moderate positive correlations (0.3~0.4) 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 better utilize L2 throughput when the locality is captured at the L2 level.

By further examining the assembly code, we observed that the compilers apply unrolling in some cases. Sometimes they may even fully unroll a loop when the trip count is relatively low,

potentially exploding the code size. Yet, the compilers fails to apply unrolling in many cases in which it would be beneficial. In particular, compilers may not unroll all loops with small bodies. Unrolling is particularly effective for these loops because they have high overhead due to loop bookkeeping. In addition, since the operation counts are low for a small loop body, unrolling does not excessively inflate the code size. It is surprising that the compilers decide not to unroll these loops, so there is a clear need to improve the profitability model for unrolling by significantly bias towards loops with small bodies.

Unrolling occasionally facilitates vectorization; however, in some cases it can also prevent vectorization. Vectorization is traditionally done by strip-mining the inner loop by the length of the vector registers and then replacing the original loop body with a vector equivalent. This means that fully unrolled loops cannot be vectorized and that partially unrolled loops could lead to inefficient vectorization. For example, ICC and Clang are both able to vectorize a loop from *SPEC 2006* by default, but unrolling the loop twice causes the compilers to produce scalar code, which leads to 12x and 14x slowdown respectively. In fact, Maleki et al. [2011] suggests that compilers sometimes re-roll a source-level unrolled loop in order to vectorize it. However, the newer basic block vectorization techniques benefit from unrolling. This technique first unroll the loop by a certain factor and then try to assemble isomorphic statements (statements that 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) by Larsen and Amarasinghe [2000].

```
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 3. Original and unrolled *TSVC s281* code

Compared with strip-mining based vectorization, basic block vectorization can partially vectorize a loop more easily. 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 3, because it exhibits the interesting trait of partial vectorization. The original loop is not well-optimized by any of the the three compilers that we evaluated. After unrolling the loop 8 times, Clang 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 dependences. Since operations on `a[]` have loop carried dependences, they remain scalar. By partially vectorizing this loop, Clang gains a 1.7x speedup. While compilers have implemented basic block vectorization, this case demonstrates that they may miss vectorization opportunities until the loops are manually unrolled.

Because unrolling can impact vectorization both negatively and positively, it is difficult for programmers and source-to-source optimizers to predict its effect on performance. To avoid this inconsistency, we propose that compilers could first determine whether or not the source code of

the loop is unrolled and then choose the appropriate vectorization pass. Alternatively, compilers could apply an unrolling or a re-rolling pass before the basic block based or strip-mining based vectorization pass respectively, and if the vectorization pass does not succeed, discard the re-rolled/unrolled results (if they are not profitable by themselves).

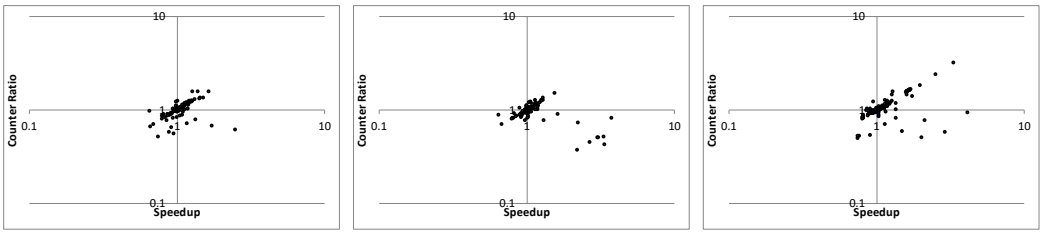
5.4 Unroll-and-Jam

Unroll-and-jam is primarily employed to facilitate data reuse by improving register usage, which decreases the number of memory accesses. In addition, it may enable vectorization on the outer loop without performing interchange [Nuzman and Zaks 2008]. Finally, while not as important, unroll-and-jam may reduce control overhead and/or increase ILP similarly to unrolling. Compilers may reverse a source level unroll-and-jam by applying an interchange → re-rolling → interchange sequence accordingly.

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`, indicating that an improvement in L1 hit rate is a major factor for the performance gain. GCC and Clang, on the other hand, have lower negative correlations (-0.6∼-0.5) with `l1d.replacement`; however, they also have negative correlations (-0.6∼-0.5) 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. In fact, 75% of ICC's data points land in quadrant 1 and 3, suggesting a positive correlation. However, for high speedup cases, ICC shows decrease in `l2_rw_rate`, represented by the points in quadrant 4. Since the Pearson correlation biases towards data points with higher values, the correlation coefficient becomes negative. In order to understand ICC's opposite correlation with `l2_rw_rate` at different speedup ranges, we inspected other metrics and found that:

(1) At high speedup range, `l1d.replacement` is significantly reduced, which means that the speedup is achieved from better data reuse and thus fewer L1 eviction.
(2) At low speedup/slowdown range, the performance change is due to other factors such as lower control overhead and/or higher ILP.



(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 quadrant 4 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. To further analyze how vectorization is affected, we define *dynamic vectorization rate* as:

$$dynamic\ vectorization\ rate = \frac{dynamic\ vector\ instruction\ count}{dynamic\ instruction\ count} \tag{6}$$

We see a general increase in vectorization rate. 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 improves the vectorization rate for 8 loops.

While unroll-and-jam helps ICC's performance, we also observed that it seemingly reduces the effectiveness of vectorization. Of the 28 loops where unroll-and-jam is beneficial, 4 loops are not vectorized initially, and the transformation does not help ICC succeed in vectorizing any of them. Instead, there are 5 loops that are vectorized initially but become not vectorized after unroll-and-jam. 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%. Two major factors contribute to this situation. First, the benefit from vectorization is overshadowed 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, unroll-and-jam may eliminate performance unfriendly patterns from the vectorized baseline, such as gather-scatter, and generate more efficient vector code, despite having lower vectorization rates.

```
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 4. Original and unroll-and-jammed *Polybench linear-algebra-blas-gemver* code

Listing 4 contains a loop nest from *Polybench*'s linear algebra workload *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 manner. 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.

5.5 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, which improves performance. Unlike the other transformations, it is harder for general purpose compilers to un-tile a loop nest.

Due to the limitations described in Section 2.2, we are only able to examine the effects of 1-D tiling. The correlation results 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 stems from the additional address calculations and iterator operations added by the new loop nest level. Therefore, tiling can cause slowdowns if the access patterns do not benefit from tiling or the benefits do not outweigh the overhead.

Furthermore, we found rare cases where tiling enables vectorization for loops with variable bounds. After tiling or strip-mining, the tiled innermost loop no longer has variable bounds. This helps the compilers whose vectorization model is confused by the original variable bounds.

5.6 Distribution

Loop distribution separates data streams, which may improve locality and/or prefetching behavior. Compilers may reverse a source level distribution by applying loop fusion. The correlation results confirm its utility. All three compilers have high positive correlations ($0.7 \sim 0.9$) with `l2_rw_rate`, indicating that distributed code may utilize L2 throughput better, 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.

6 VECTORIZATION

In Section 5, we have already discussed that vectorization plays a major role in the performance variation. If a loop is not originally vectorizable but, after undergoing a transformation sequence, becomes vectorized, it may receive a sizable performance boost. On the other hand, a loop's performance may suffer greatly if a transformation sequence disables vectorization. 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. Such performance variation caused by vectorization contributes to compilers' instability significantly. Therefore, we took one step further to investigate how different vectorization settings may influence the performance of loops.

We compiled and profiled the loop nests and their mutations with 4 more vectorization settings. We refer to the compiler settings described in Section 3 as the reference settings, and the additional settings are the reference settings with added switches. They are: generating scalar code only,

using only SSE, using SSE and AVX, and using up to AVX2. For the three vector configurations, we disabled the compilers' vectorization profitability analysis if possible so that the compilers vectorize a loop with the corresponding vector extension whenever possible, regardless of the predicted profitability. Note that Clang 4.0.0 does not provide switches to turn off its profitability analysis.

Because vectorization does not guarantee speedup, instead of looking at a compiler's vectorization report to determine whether a loop is vectorized, we use the method from Maleki et al. [2011] and define that a loop has *effective* vectorization if the vector code is at least 15% faster than the scalar code; specifically, we claim that a vectorization attempt 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.

6.1 Effect of Transformations on Vectorization

Let's first investigate how source-level transformations affect vectorization. The first row of Table 8 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%~47.5%) are much higher than the percentages of loops with beneficial mutation in L, which are 25.9%~36.6% (second row in Table 3). 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%~38.1% of the beneficial mutations are vectorized effectively while their baselines are not; thus, applying the correct source-level transformations can boost compilers' vectorization success rate.

Table 8. Statistics of effective vectorization

		GCC	ICC	Clang
1	# of loops studied (L)	1241	1175	1266
2	# (%) in L without effective vectorized baseline (N)	866 (69.8%)	700 (59.6%)	999 (78.9%)
3	# (%) in N that has beneficial mutation (B)	373 (43.1%)	277 (39.6%)	475 (47.5%)
4	# (%) in B whose baseline is not vectorized but has vectorized beneficial mutation(s)	141 (37.8%)	100 (36.1%)	181 (38.1%)

6.2 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 9 contains the number of loops that are improved by at least 15% via changing vectorization settings. Row 2 to 4 focus on the benefit by bypassing the profitability model. Row 2 shows that 6.0%~8.1% of the loops can be sped up over 15% by forcing the compilers to yield scalar

code. This situation occurs when the overhead introduced by vectorization (e.g. gather/scatter) is higher than the benefit of vectorization yet the profitability model fails to assess it. Row 3 is for the opposite scenario when the loop is vectorizable and profitable, but the profitability model deems otherwise. Because we were not able to turn off Clang's vectorization profitability model, only GCC and ICC produce this situation, and 2.6%~3.1% of the loops fall in this category for them. Row 4 is the subtotal of the above two situations. Row 5 counts the loops whose performances increase when vectorized with an older vector extension, i.e. SSE or AVX. In this category, vectorization profitability model is disabled for GCC and ICC but enabled for Clang. This time, 10.0%~12.1% additional loops receive benefit. Finally as shown in the last row, 18.1%~21.5% of the loops can receive a sizable performance boost by changing vectorization settings only and without undergoing any transformation. Note that although Clang has the lowest value in the last row, its vectorization profitability model is not necessarily better than those of the other two compilers; in fact, Clang having the highest value in row 2 suggests otherwise. We believe Clang's low percentage in the last row heavily attributes to the inability to disable its profitability model.

Table 9. Statistics of loops having speedup by changing vectorization settings

		GCC	ICC	Clang
1	# of loops studied (L)	1241	1175	1266
2	# (%) in L best improved with forced scalar	85 (6.8%)	71 (6.0%)	102 (8.1%)
3	# (%) in L best improved with forced vectorization	32 (2.6%)	36 (3.1%)	N/A
4	Subtotal	117 (9.4%)	107 (9.1%)	102 (8.1%)
5	# (%) in L best improved with older vector ext.	150 (12.1%)	138 (11.7%)	127 (10.0%)
6	Total	267 (21.5%)	245 (20.9%)	229 (18.1%)

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.

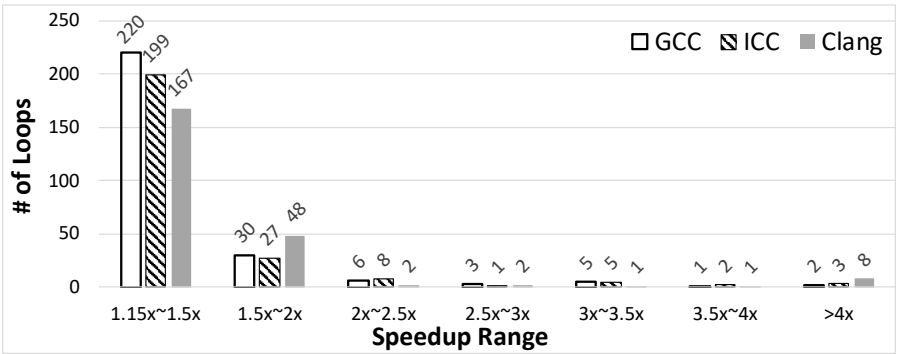


Fig. 7. Distribution of loops that are improved by changing vectorization settings when compiling the original loop

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 by over 15% for GCC, ICC, and Clang respectively, and the average speedup of these beneficial cases is 1.61x~1.65x depending on the compiler. The average lower bound of performance headroom by applying source-to-source transformations as well as searching for the best vectorization setting is 23.7% for GCC, 18.1% for ICC, and 26.4% for Clang.

6.3 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 5, which is taken from NPB LU benchmark and whose scalar version outperforms its vectorized counterparts. ICC's vectorized reference compilation of the codelet is 2.2x slower than the scalar compilation. After investigating the assembly code, 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 5. 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 vector register, the compiler unrolls the loop by a factor of 2 and then picks two adjacent elements in a column, e.g. `ce[0][0]` and `ce[1][0]` to form the vector. Since one vector instruction can process operations from multiple loop iterations in the source code, one would expect the vector code's assembly loop trip count to be less than that of the scalar one. However, the trip count of the scalar loop is instead half of that of the vectorized loop. 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 enhances instruction pipelining with the help of better ILP.

Second, this loop nest 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 The 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 time 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 time while the vector code keeps the memory busy for only 31% of the execution time. Since the most expensive factor is write cache misses, and the scalar version manages to process it 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, causing the processor to stalls and prevent the processor from executing the stores more aggressively.

Consequently, the fact that compilers may fail to accurately predict the outcome of vectorization due to complex factors interfering with each other reduces the compilers' stability.

7 RELATED WORK

Maleki et al. [2011] 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. They applied transformations on a smaller collection of loop nests by hand and focused only on vectorization. In comparison, we conducted our study on a large number of programmatically generated mutations from an extensive collection of loop nests, and we also investigate other components of the optimization process besides vectorization. Most importantly, the main perspective that our paper studies: stability, is not explicitly considered by them.

Exploring the space of program variants as a mechanism to test for correctness of compilers has been studied extensively. A variant is Equivalence Modulo Inputs (EMI) by Le et al. [2014], which transforms source programs to generate versions that are semantically-equivalent not for all inputs but for a specific set of program inputs. A compiler defect is detected when the target code from two different versions produce different outputs. The *GLFuzz* technique by Donaldson et al. [2017] is similar to our approach in that it applies semantic-preserving transformations to check the correctness of GLSL compilers. Other similar techniques to find performance bugs is discussed by Segura et al. [2017]. They call the strategy *Performance Metamorphic Testing*. The work reported in this paper can be considered as a member of this class of performance testers.

Park et al. [2013] used the performance counter values gathered from executing a loop as the input to a machine learning model that predicts the best polyhedral transformations for the loop. We instead use the correlation between performance counters and performance to investigate the effect of source-level transformation.

Source-to-source transformations have been used as a compiler pre-pass in prior researches in order to improve code performance. Tiwari et al. [2011], Fursin et al. [2002], and Pouchet et al. [2008] adopted the technique in iterative compilation. Tiwari et al. [2011] used the *CHiLL* framework [Chen et al. 2008] to perform source-level transformations, and they pointed out that the transformation search space grows exponential in size as the number of tuning parameters increases. Fursin et al. [2002] searched the transformation space that consists of three transformations: array padding, loop unrolling, and loop tiling. Pouchet et al. [2008] searched polyhedral transformation space in their iterative optimization approach. Polyhedral compilers such as *PLUTO* [Bondhugula et al. 2008] and the work from Adamski et al. [2016] also work as a source-to-source pre-pass to the back-end compiler. The polyhedral transformations that they apply can be viewed as sequences to loop transformations. However, both iterative compilation and polyhedral compilation only aim to find a good shape of a given loop nest within reasonable time. Therefore, iterative compilation uses search algorithms to converge to high performance within limited number of steps, and polyhedral compilation applies a single compound transformation based on the polyhedral model. On the other hand, our work purposefully cover a large transformation space in order to evaluate the stability of compilers and to understand the effect of source-level transformations.

Aimed at accelerating performance evaluation of programs, a few prior works also proposed to extract the hotspots from applications and save them as stand-alone codelets. Castro et al. [2015] 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. [2009] and Tiwari et al. [2011] 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.

8 CONCLUSION

This paper presents the first quantitative study on compiler stability – the measure of the variation in performance of the target code generated by a compiler from semantically equivalent yet differently structured source code. In this study, we investigated the stability of GCC, ICC, and Clang's compilation processes of C language `for` loop nests. In addition, we also estimated the performance headroom of the said processes.

We measured the compilers' stability and performance headroom by profiling an extensive collection of loop nests extracted from benchmarks and libraries along with their semantically equivalent mutations generated by applying sequences of semantic-preserving transformations.

We quantified compiler stability by introducing a pair of *stability scores*. The *intra-compiler stability score* measures the average variation in performance of semantically equivalent mutations compiled by a given compiler. The score indicates that the three studied compilers are far from being stable. The *inter-compiler stability score* measures the average variation in performance of the target code generated by multiple compilers from the same loop semantics. The score reveals a noticeable performance gap among the compilers. We also used the score to confirm that source-to-source transformations are able to narrow the gap. We believe that the two stability scores are useful tools for compiler developers to evaluate the stability of their compilers as well as for the community to track the progress to compiler stability over time.

To understand the reasons that cause the instability in the compilation process, we analyzed the major effects of the transformations on performance by correlating the performance variation with the change in performance counter readings and derived metrics. Using the correlation and by manually inspecting the assembly code of interesting cases, we were able to suggest improvements on compiler design that may increase the compilers' stability. We also found that the effect of source-level transformations is sometimes difficult to predict. For example, unrolling may either help or harm vectorization depending on the vectorization technique employed. 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.

Because vectorization has a significant impact on performance and stability, we also investigated how different loop structures affect vectorization as well as how effective the compilers' vectorization profitability models are. We observed that when the vectorization profitability model fails, the performance of a loop can be severely harmed. Also, the newest vector extension, despite having longer vector length and more features than the older ones, can be outperformed by the older ones.

With the combined effort of applying source-to-source transformations and tuning vectorization settings, 35.7~46.5% of the loops, depending on the compiler, exhibit a performance headroom of over 15%. Depending on the compiler, the average performance headroom of these significantly improved loops is 61.4%~65.3%, and the average performance headroom of all studied loops is 18.1%~26.4%. The results are the lower bound of potential performance improvement. We believe that the actual headroom may be significantly higher. By expanding the experiment with more loops and transformations in the future, we can gradually raise the lower bound and eventually get a sense of the actual performance headroom and instability of the compilers.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Award 1533912 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=1533912). We thank Gerald DeJong for his valuable inputs during the early stage of the project. We thank the OOPSLA 2018 reviewers for their valuable comments and recommendations.

REFERENCES

Dominik Adamski, Grzegorz Jabłoński, Piotr Perek, and Andrzej Napieralski. 2016. Polyhedral Source-to-Source Compiler. In *Mixed Design of Integrated Circuits and Systems, 2016 MIXDES-23rd International Conference*. IEEE, 458–463.

Randy Allen and Ken Kennedy. 2001. Optimizing compilers for modern architectures a dependence-based approach. (2001).

David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications* 5, 3 (1991), 63–73.

C Bastoul and LN Pouchet. 2012. *Candl: the chunky analyzer for dependences in loops.* Technical Report. tech. rep., LRI, Paris-Sud University, France.

Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. 2008. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*. Springer, 132–146.

Shirley Browne, Jack Dongarra, Eric Grosse, and Tom Rowan. 1995. The Netlib mathematical software repository. *D-Lib Magazine, Sep* (1995).

Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. 2015. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Trans. Archit. Code Optim.* 12, 1 (2015), 6:1–6:24.

Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. Citeseer.

Zhi Chen, Zhangxiaowen Gong, Justin Szaday, David C. Wong, David Padua, Alexandru Nicolau, Alexander V. Veidenbaum, Neftali Watkinson, Zehra Sura, Saeed Maleki, Josep Torrellas, and Gerald DeJong. 2017. LORE: A loop repository for the evaluation of compilers. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*. 219–228.

Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *PACMPL* 1, OOPSLA (2017), 93:1–93:29. https://doi.org/10.1145/3133917

Stuart I Feldman. 1990. A Fortran to C converter. In *ACM SIGPLAN Fortran Forum*, Vol. 9. ACM, 21–22.

Jason E Fritts, Frederick W Steiling, Joseph A Tucek, and Wayne Wolf. 2009. MediaBench II video: Expediting the next generation of video systems research. *Microprocessors and Microsystems* 33, 4 (2009), 301–318.

GG Fursin, Michael FP OâĂŹBoyle, and Peter MW Knijnenburg. 2002. Evaluating iterative compilation. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 362–376.

GAP. 2007. GAP - Groups, Algorithms, Programming - a System for Computational Discrete Algebra. www.gap-system.org.

John L Henning. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer* 33, 7 (2000), 28–35.

John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

Intel. 2016. Intel 64, and IA32 architectures software developer's manual, vol. 3A: system programming guide, part 1. *Intel Corporation, Denver, CO* (2016).

David J. Kuck, Robert H. Kuhn, Bruce Leasure, and Michael Wolfe. 1980. The structure of an advanced vectorizer for pipelined processors. In *Fourth International Computer Software and Applications Conference*. IEEE, 201–218.

LAME. 2017. LAME MP3 Encoder. lame.sourceforge.net.

Samuel Larsen and Saman Amarasinghe. 2000. *Exploiting superword level parallelism with multimedia instruction sets*. Vol. 35. ACM.

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 216–226. https://doi.org/10.1145/2594291.2594334

Man-Lap Li, R. Sasanka, S. V. Adve, Yen-Kuang Chen, and E. Debes. 2005. The ALPBench benchmark suite for complex multimedia applications. In *IEEE International Proceedings of the IEEE Workload Characterization Symposium (IISWC)*. 34–45.

Chunhua Liao, Daniel J Quinlan, Richard Vuduc, and Thomas Panas. 2009. Effective source-to-source outlining to support whole program empirical optimization. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 308–322.

LLNL. 2008. ASC Sequoia Benchmark. https://asc.llnl.gov/sequoia/benchmarks/.

Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. 2011. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 372–382.

Mozilla. 2017. Mozilla JPEG Encoder Project. github.com/mozilla/mozjpeg.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. 2009. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices* 44, 3 (2009), 265–276.

Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization-revisited for short SIMD architectures. In *Parallel Architectures and Compilation Techniques (PACT), 2008 International Conference on*. IEEE, 2–11.

Zhelong Pan and Rudolf Eigenmann. 2006. Fast, Automatic, Procedure-level Performance Tuning. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 173–181.

Gabriele Paoloni. 2010. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation* (2010), 123.

Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and P Sadayappan. 2013. Predictive modeling in a polyhedral optimization space. *International journal of parallel programming* 41, 5 (2013), 704–750.

Tim Peters. 1992. Livermore loops coded in C. http://www.netlib.org/benchmark/livermorec. (1992).

Louis-Noël Pouchet. 2011. Polyopt/C: A polyhedral optimizer for the ROSE compiler. http://web.cse.ohio-state.edu/~pouchet/software/polyopt.

Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. http://www.cs.ucla.edu/pouchet/software/polybench. (2012).

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. 2008. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 90–100.

Dan Quinlan. 2000. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters* 10, 02n03 (2000), 215–226.

Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

Peter Rundberg and Fredrik Warg. 2002. The FreeBench v1.0 Benchmark Suite. http://www.freebench.org. (2002).

Sergio Segura, Javier Troya, Amador Durán Toro, and Antonio Ruiz Cortés. 2017. Performance Metamorphic Testing: Motivation and Challenges. In *39th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track, ICSE-NIER 2017, Buenos Aires, Argentina, May 20-28, 2017*. 7–10. https://doi.org/10.1109/ICSE-NIER.2017.16

Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. 2014. CortexSuite: A Synthetic Brain Benchmark Suite. In *IEEE International Proceedings of the IEEE Workload Characterization Symposium (IISWC)*. 76–79.

Ananta Tiwari, Jeffrey K Hollingsworth, Chun Chen, Mary Hall, Chunhua Liao, Daniel J Quinlan, and Jacqueline Chame. 2011. Auto-tuning full applications: A case study. *The International Journal of High Performance Computing Applications* 25, 3 (2011), 286–294.

Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. 2003. Compiler Optimization-space Exploration. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*. 204–215.

TwoLAME. 2017. TwoLAME - MPEG Audio Layer 2 Encoder. www.twolame.org.

Rob F Van der Wijngaart and Timothy G Mattson. 2014. The Parallel Research Kernels.. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6.

Jeffrey S. Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (March 1985), 37–57.

xiph.org. 2017. Codecs from Xiph.Org Foundation. https://www.xiph.org.