LORE: A Loop Repository for the Evaluation of Compilers

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

***Abstract*—Although numerous loop optimization techniques have been designed and deployed in commercial compilers in the past, virtually no common experimental infrastructure nor repository exists to help the compiler community evaluate the effectiveness of these techniques.**

This paper describes a repository, LORE, that maintains a large number of C language `for` loop nests extracted from popular benchmarks, libraries, and real applications. It also describes the infrastructure that builds and maintains the repository. Each loop nest in the repository has been compiled, transformed, executed, and measured independently. These loops cover a variety of properties that can be used by the compiler community to evaluate loop optimizations using a broad and representative collection of loops.

To illustrate the usefulness of the repository, we also present two example applications. One is assessing the capabilities of the auto-vectorization features of three widely used compilers. The other is measuring the performance difference of a compiler across different versions. These applications prove that the repository is valuable for identifying the strengths and weaknesses of a compiler and for quantitatively measuring the evolution of a compiler.

I. Introduction

A significant fraction of execution time is consumed by loops for a large class of programs. For this reason, numerous loop optimization techniques [1], [2] have been designed and deployed in all major compilers. These optimizations transform loops into semantically equivalent versions that exhibit better locality, require less computation, and/or take advantage of parallelism from vector devices/multicores. Furthermore, compiler optimizations also enhance programability and enable machine independence. The ultimate goal of research in automatic program optimization is to enable programmers to focus on the preparations of readable and correct code because they can rely on compilers to generate highly efficient code for each target machine. It is well known that we are far from reaching the ultimate goal and that compilers today are brittle in that it is impossible to know before hand if they will be able to generate a highly efficient version of the code or if, on the contrary, it will be necessary to devote considerable time to manually transform the code for good performance.

Clearly, there is much room for advances in compiler technology. An important enabler of these future advances is data on the effectiveness of compilers on different classes of codes and target machines, the progress of compilers across successive versions, and the effectiveness of solo/compound transformations. While some of such data can be found in the literature, in many cases compiler studies are confined to a few codes, which are not always widely available. In addition, there is little in the area of historical data that shows how much progress compiler technology has made in terms of delivering performance.

In this paper, we propose *LORE*, a repository of program segments, their semantically equivalent transformed versions, and performance measurements with various compilers. Our goal is to create an extensible repository for the compiler community that can provide a common base for experimentation and comparison of results, allowing compiler writers to see the effect of transformations on individual loop nests and to compare execution times of different transformation sequences. Currently, the repository contains:

- A growing number of C language `for` loop nests (∼2,500 as of the writing of this paper) extracted from multiple benchmarks, libraries, and real applications by an *extractor*. The extractor encapsulates loop nests into standalone executables called *codelets*. Each codelet executes a loop using data captured during the original benchmark execution, measures its execution time, and collects readouts from hardware performance counters for further analysis.
- Multiple semantically equivalent versions of each loop produced by a source-to-source *mutator* that applies sequences of loop transformations constructed by combining tiling, interchange, distribution, unrolling, and unroll-and-jam into sets of possibly repeated subsets of these transformations. We call the result of applying each of the transformation sequences a *mutation*. The repository currently contains ∼90,000 different mutations (an average of 36 mutations per original loop), as of the writing of this paper.
- A database that correlates the execution profiling of the original loop and its mutations with the compilers and target machines used for each execution. This data is valuable not only for comparing the effectiveness of different compilers' optimization passes but also for tracking the evolution of a compiler's capability in loop optimization. Currently the repository contains data on the execution on a single target machine from 2 versions of the *Intel® C++ Compiler* (ICC), the *GNU C Compiler* (GCC), and *Clang* (frontend of LLVM for C family languages).

We extract the loops from the benchmarks for two reasons. One is to reduce the time to evaluate each loop. Evaluating the effect of different compilers with various switches on the original loop nests and their semantically equivalent mutations typically requires numerous executions of the loop. Much time is saved by executing the loop and its mutations in isolation. The second reason is that it is important to study the execution of the loops separately so that we could classify loops for the purposes of applying machine learning techniques for compilation and, when necessary, alter the context of execution of each loop.

Both the source code for the original loop and its mutations and the measurement data are available for download/query from the *LORE* website. The web interface also conveniently allows users to view dynamically generated plots/charts from selected loops/compilers.

In order to illustrate the usefulness of *LORE*, we include in this paper two experiments. One compares the effectiveness of major compilers' auto-vectorizers since vectorization plays an important role in performance improvement and efficient hardware utilization. The other assesses the performance difference across generations of a compiler. These experiments prove that *LORE* is an efficient approach to effectively identify the strengths and weaknesses of a compiler and to quantitatively measure the evolution of a compiler.

The remainder of the paper is organized as follows. Section II presents the tools contained in the infrastructure for loop extraction, mutation, and clustering. We explain the details of the repository in Section III. Section IV gives two example analysis that users can perform using the data in the repository. Section V discusses related work and compares our repository and infrastructure tools with the state-of-the-art research. Finally, Section VI concludes the paper.

II. Modules to create and access *LORE*

Figure 1 shows the different software modules that we have built to create and access *LORE*, which include:

- an extractor to create codelets,
- a mutator to create multiple semantically equivalent versions (mutations) of each codelet,
- a clusterer to classify the codelets,
- a database as the repository to store various measurements of the codelets/mutations,
- a web interface to give broad access to the repository.

In this section, we describe these modules. The first three were implemented using the ROSE source-to-source compiler infrastructure[3] and operate on C programs, one of the most widely used languages for high-performance computing.

A. Extractor

The extractor separates loops from benchmark programs to create codelets. There are three steps in the separation process. First, the extractor traverses the *abstract syntax tree* (AST) and identifies the `for` loops to be extracted. Second, the extractor copies and makes available to each of the codelets the value of all data that the codelet's loop accesses. That is, all the

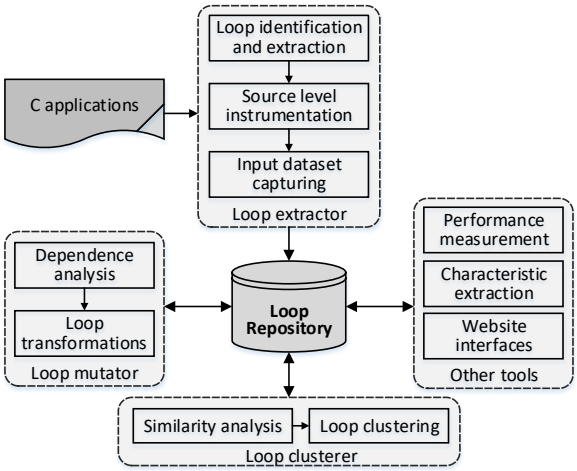


Fig. 1: The system framework of the infrastructure.

data items that the loop reads before being modified during its execution. This is done to guarantee that the execution of the codelet will follow the same flow of control as the benchmark. The memory layout is preserved for data accessed in the loop, however the extractor does not replicate the cache state when a benchmark executes the target loop since cache state is invisible to the programmers during coding. Finally, the extractor copies the loop statements, creates the necessary declaration, and inserts the instrumentation statements.

Data copying. For each `for` loop nest to be separated, the extractor inserts code into the source file to first capture the data read inside the loop and then save them to a file. To this end, the loop nest is analyzed statically to identify all variables that are read-only or read-before-write. Variables that are guaranteed to be write-only or write-before-read are not instrumented as their values are generated during the execution of the loop. Then, for these collected variables the values are obtained as follows depending on the nature of the variables.

- Non-pointer scalar variables are instrumented simply by adding code to write their values to a file directly.
- Arrays with known dimensions are treated similarly to scalar variables. Statements are added before the target loop to write the values of all array elements to a file in a row-major way.
- Pointers and arrays of unknown sizes (e.g. `int a[]`) are handled following a two-pass procedure. In the first pass, all referenced memory locations are captured so that the memory boundaries in static, heap, and stack sections accessed by the loop are determined. In the second pass, the benchmark is executed again, and all the memory contents between the lower and upper boundaries of each memory section is saved to a file. To guarantee that the same pointer would have fixed addresses during the initial extraction and the later codelet execution, we disable the *address space layout randomization* (ASLR).
- The address of a structure variable is logged if it is a pointer. Otherwise, it is treated as a basic scalar type.

A target loop may be executed many times, i.e. it can be in a function called by another function and the caller is in a loop. To reduce the prohibitively high performance

and space overhead caused by instrumentation, our system currently saves data for only one invocation of the loop nest by using a sampling technique called reservoir sampling [4]. The sampling method guarantees that each execution instance is equally likely to be selected. This approach can be easily extended to enable the extraction of data for multiple executions of a loop.

Loop Statement Extraction. The third pass of the extractor copies the statements of the target `for` loop (nest) into a separate file to form a codelet. The codelet can then be compiled, transformed, executed, and measured as a standalone program. This pass creates the codelet by inserting into it:

- the executable statements of the target loop,
- the declaration of each variable used by the loop,
- the user-defined types that apply to variables referenced by the loop nest, e.g. enumerations, structures, and `typedef`,
- statements to load the data collected from instrumentation and statements to assign these values to variable in the program,
- statements to measure execution time and to read performance counters.

The first three tasks are relatively easy. Thus, in ROSE, we traverse the AST to analyze the type of each variable in the loop. When creating the declaration of a variable, we first check if it is a `C` built-in type or a user-defined type. `C` built-in types can be ignored, but user-defined types have to be handled carefully. We recursively extract all the directly and indirectly used user-defined type declaration chains before creating the declaration.

After collecting all declarations, we initialize them according to their data types as follows.

- Non-pointer scalar variables are initialized using the value collected from instrumentation directly.
- Each element of an array with known size is assigned with the value saved in the data file.
- Arrays of unknown sizes are created as pointers, i.e. `int a[]` is declared as `int *a` in the extracted loop. We fill the related regions of global, heap, and stack sections with the copied data so that the isolated loop has the same memory mapping as the original loop. Each pointer, including the one declared from an unknown size array, is assigned the recorded memory location that is associated with the pointer in the original program. The current stack is saved and restored before and after each execution of the loop to guarantee the correct execution of the isolated loop.

B. Mutator

The mutator applies to a codelet sequences of source-to-source, semantically-preserving transformations and creates one codelet for each loop version or mutation. As of the writing of this paper, the transformation sequences components are: interchange, tiling, unrolling, unroll-and-jam, and distribution. These five are among the most basic and widely used transformations. Dependence analysis is used to identify which transformations can be applied. As shown in Table I all transformations are parameterized except for distribution, which is applied to maximize the number of resulting loops.

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

TABLE I: Transformations and their parameters

To reduce the potentially immense number of mutations, the current version of the mutator:

- only unrolls the innermost loop.
- tiling is applied to a single loop level in each transformation sequence.
- only the innermost loop is (fully) distributed.

Unlike unrolling, which is always legal, the application of the other transformations is controlled by dependence analysis carried out by *PolyOpt/C* [5]. The main limitation is that *PolyOpt/C* is based on the polyhedral model, which only accepts loops with affine loop bounds and array subscripts. For example, loops with array access like `A[i*j]` or `A[B[i]]`, or with loop bound test like `j < i * i`, are not amenable to the polyhedral model. Consequently, currently only unrolling is applied to non-affine loops.

The mutator is currently able to apply interchange, tiling, and unroll-and-jam only to perfect loop nests. As a result, these transformations could not be applied after any transformation that may render a loop nest imperfect (e.g. unroll-and-jam may produce a residue loop). In addition, to keep the total number of mutations from thousands of loop nests reasonable, the mutator does not explore the transformation space exhaustively; instead, it transforms the loop nests in a selection of orders. Currently, the possible orders are subsets of:

$$interchange \rightarrow unroll\text{-}and\text{-}jam \rightarrow distribution \rightarrow unrolling \quad (1)$$

$$interchange \rightarrow tiling \rightarrow distribution \rightarrow unrolling \quad (2)$$

e.g. $interchange \rightarrow distribution$ is a plausible transformation sequence, but $distribution \rightarrow interchange$ is not.

One could argue that modern compilers would be able to provide the same performance if they could provide accurate enough dependence analysis and sufficient transformations. However, we have found that using the mutator as a pre-pass in many cases improves performance. This means that there is much room for improvement of modern compilers. Indeed, application of the mutator reveals the weaknesses of analysis of today's compilers and serves as a guide to identify the most performance critical transformation (sequence).

C. Clusterer

Similar loops that share the same series of optimal transformations will be be clustered together to reduce the overhead required to include new programs in our repository. If a new loop fits into a cluster, the time consuming mutation generation process required to analyze it could be skipped

altogether. Furthermore, the size of each cluster is an effective indicator of how common the corresponding loop pattern is. Compiler developers may use such information to prioritize the optimization for common loop patterns.

The clusterer will make use of a static analysis process to identify features of a newly submitted loop and find loops in the repository with common features. Then, these loops from the repository will be compared against the test loop via a semantics test to generate a similarity score. Loops with a sufficiently high similarity score can then be clustered together. These tests would be specifically tuned for our application; for example, we would consider the statements `a[i] = 2 * b[i] + 1` and `a[i] = 3 + 2 * b[i]` to be similar since their behavior under different transformation techniques are expected to be identical.

D. Website

A web-based user front-end[1] is designed for public accesses and queries, allowing compiler and language researchers to not only search the repository but also contribute new benchmarks and measurements. The website provides the following functionalities:

- allows download for source code and compiled assembly of the codelets and their mutations;
- plots experimental results at different granularity levels (loop level, application level, benchmark level, etc.);
- presents static/dynamic features of the codelets in searchable/sortable interactive tables;
- allows users to run read-only SQL queries.
- lets users to submit comment and analysis for each codelet.

New features such as new benchmark submission system are also being developed.

III. Repository: LORE

The main objective of the tools described in Section II is to add data to the *LORE* repository. The repository contains the codelets, the mutations created by the mutator, and their main dynamic and static characteristics. The repository also contains information about execution times and the context in which these times were measured including the target machine, compiler, and compiler switches.

In this section, we discuss the underlying database used for the repository, the sources from where we have extracted loops so far, the main characteristics that we measure for each loop, and the context in which we have carried out measurements.

A. Database

The core of the repository is a MySQL database that holds specific information such as:

- the source of each loop (benchmark name, benchmark version application name, file name, function name, line number, etc.)
- the static/dynamic features of the extracted loops (described below in Section III-D)
- the transformation sequence involved in a mutation and their parameters
- details of each experimental result entry, including the environment of the experiment (compiler vendor, compiler version, CPU model, etc.) and statistics of the result. (min, max, standard deviation, median, and mean of the clock cycles spent inside the loop)
- correlation metrics used for clustering loops.

The database is hosted on a dedicated server. The web server and the machines that run the measurements access the database via network.

[1]The website can be accessed at: http://www.vectorization.computer

B. Source of Loops

LORE currently contains 2499 C loops extracted from 25 widely used benchmarks, libraries implementing algorithms such as audio/video codecs and deep learning, and some real-world applications selected from GitHub. A total number of 88661 mutations have been obtained from these loops. Table II lists the number of loops obtained from each workload and the number of mutations that our mutator has successfully derived from each of them. Most of the libraries and applications were obtained from open source repositories such as GitHub and SourceForge. The libraries include daala, flac, libogg, silk, and libsndfile from Xiph.org Foundation [6], lame codec [7], twolame codec [8], and libdeep [9]. The real-world applications are GAP [10] and mozjpeg [11]. We expect the number of loops to continue to grow in size and diversity of origin.

TABLE II: The number of loop nests extracted from each benchmark and the mutation count applied on each of them.

Benchmark	# of loop nests	# of mutations
ALPBench[12]	71	384
ASC-llnl[13]	22	352
Cortexsuite[14]	111	1964
Fhourstones[15]	2	20
FreeBench[16]	71	641
Kernels[17]	164	3736
Livermore[18]	76	2404
MediaBench[19]	350	3124
Netlib[20]	44	584
NPB[21]	379	59464
Polybench[22]	92	2968
Scimark2[23]	3	24
SPEC[24]	665	8089
TSVC[25]	152	2012
libraries	242	2513
real-world apps	55	382
Subtotal	2499	88661

C. Codelets

As described in Section II-A, the extractor identifies `for` loops in a program and creates a codelet for each of them. A codelet consists of two separate files. One is a new source file which only contains the loop nest copied from the original application, declaration of all variables used by the loop, and user defined data types (e.g. `typedef`, `struct`, and `enum`, etc). The other one is composed of four primary parts: 1) the loop invocation handler, i.e. we may need to execute the standalone loop multiple times, 2) operations to fetch

data that is required by the loop nest from the file built by instrumentation, e.g. initializing variables with either values or pointer addresses, 3) timers to measure the number of machine cycles executed by a loop, 4) APIs to access the performance counter handling module. Other operations such as memory allocation and deallocation, saving and restoring stack contents, read/write machine specific registers (MSRs), and statistical analysis of execution time results are programmed into shared libraries that can be linked to a codelet without recompilation.

We use the `RDTSCP` instruction to read the CPU's timestamp counter (TSC) for measuring the number of clock cycles consumed by running an extracted loop. This instruction guarantees that all instructions before it have retired from the pipeline before it reads the TSC; therefore, it is able to accurately gauge loops with small execution cycles. Each codelet can be specified to execute multiple times by giving a parameter. The minimum, maximum, mean, median, and standard deviation of the cycle counts are calculated at the completion of all runs.

Although a codelet is independent of the original program, it still uses the same input data and has the same memory layout as the original loop. Now one can focus on the isolated code snippet rather than the full application for analysis and optimization. This facilitates applications that have well-defined hot loop kernels because compiler-based auto tuning for loop transformations could be performed offline on the codelet yet still able to deliver significant performance enhancement for the whole application.

D. Characteristics

Each codelet possesses a series of source-level static characteristics, and each measurement on a combination of compiler and machine configurations produces a list of dynamic characteristics. Currently the measured static features include:

- loop nest level,
- lower/upper bounds and stride of the loop,
- number of statements in the loop,
- number of memory operations in the loop,
- number of floating-point/integer operations in the loop,
- number of constant in the loop,
- number of branches (e.g. `if` statements) in the loop,
- number of basic blocks (BB) and edges in the loop's control flow graph (CFG),
- number of BB with different characteristics (e.g. BB with 1 predecessor and 2 successors).

When one of the static features such as loop bounds and the stride size are not available at compile time, the expression that represents the feature is stored in the repository. Note that the static-features are obtained at source-level, meaning that they may be altered by optimization passes during compilation.

Dynamic characteristics contain hardware-related activities and dynamic instruction mixes. Hardware-related activities are collected at run-time by reading hardware performance counters available on most modern microprocessors. A standalone module is designed to provide simple and high level APIs for the acquisition of performance counter values. A total of forty dynamic features have been collected for each loop nest in the repository. A subset of the performance counters include[2]:

- memory traffic, such as the number of L1D data line replacements (L1D.REPLACEMENT).
- memory hits, such as the number of retired load uops with L1/2 cache hits as data sources and misses in the L3 cache (MEM_LOAD_UOPS_RETIRED.L1/2/3_HIT_PS), etc.
- line fill buffer (LFB) occupancy, such as the number of cycles with L1D outstanding load misses (L1D_PEND_MISS. PENDING_CYCLES), etc.
- TLB related, such as DTLB_LOAD_MISSES.STLB_HIT, DTLB_STORE_MISSES.STLB_HIT, etc.
- resource stalls, such as resource-related stall cycles (RESOURCE_STALLS.ANY) and cycles stalled due to re-order buffer full (RESOURCE_STALLS.ROB), etc.
- prefetcher ralated, such as the number of L2 prefetching requests that miss the L2 cache (L2_RQSTS.L2_PF_MISS) and the number of requests from L2 hardware prefetchers (L2_RQSTS.ALL_PF), etc.

The dynamic instruction mix of each extracted loop is obtained via a customized Pin [27] tool, which collects the following statistics/instruction counts during the codelet/mutation execution:

- total number of instructions
- total number of floating point instructions
- number of floating point SSE, AVX, and AVX2 instructions
- number of integer SSE and AVX2 instructions
- number of scalar and vector loads
- number of scalar and vector stores

The static and dynamic characteristics can be used to understand performance results, determine the type of effect each mutation has over the performance of the original loops, and to serve as features in machine learning models to cluster loops and in the future to predict the best mutations for a given loop.

E. Measurements

Compiler researchers may download the codelets and their mutations from *LORE* to carry out measurements, or they may query the database to get our routinely expanded measurements. We have a number of measurements in the current version of the repository. Table III lists the compilers and their versions we have used for our measurements. In the rest of the paper, when discussing the experimental results, a compiler refers to the more recent version if we do not specify the version number, i.e. GCC means GCC 6.2.0.

GCC	ICC	Clang
4.8.5	15.0.6	3.6.2
6.2.0	17.0.1	4.0.0

TABLE III: Compilers and their versions measured and recorded in *LORE* as of Jun 2017

All experiments to this date were conducted on an Intel Haswell generation Xeon E5-1630 v3 processor (equipped

[2]Please refer to Intel Software Developer's Manual for the details of each counter [26].

with 32K L1 cache, 256K L2 cache, and 10MB L3 cache) and 32GB DDR4 2133 RAM, running Ubuntu 16.04 server version. We executed all measurements on the same CPU core with dynamic frequency scaling, Intel's TurboBoost technology, Hyper-Threading, and CPU sleep state all disabled. More experiments will be conducted with different machine settings (e.g. different cache sizes, architecture generation, memory interface speed, etc.) in the future to investigate more architectural behaviours (e.g. cache misses) of the codelets.

Each codelet and its mutations have been compiled with `-O3` for all three compilers. To allow more aggressive optimizations, we turned on additional flags for each compiler. For GCC, `-ffast-math`, `-funsafe-loop-optimizations`, and `-ftree-loop-if-convert-stores` are enabled. `-ffast-math` is also enabled for Clang. `-restrict` is used for ICC compilation to promote pointer aliasing analysis. We call the executable compiled with the above flags the *reference* compilation.

In order to assess compilers' vectorization capability with different generations of Intel's vector extensions, we also compiled each codelet/mutation with 4 additional settings besides the reference: only allow SSE vector extension, allow up to AVX vector extension, allow up to AVX2 vector extension, and only allow scalar instructions. To force the compilers to apply these settings, we disabled compilers' vectorization profitability model if possible.

The performance statistic results are collected over 100 runs of each codelet with the above compiler flags. The original code with the above flags is used as the *baseline* for comparisons, and the one among all mutations that produces best performance on a compiler with the same flags is referred to as the *best mutation*.

IV. Example Analysis

This section describes two example studies based on the loop nests in LORE. The first one compares the capability of three production compilers' auto-vectorizers. This experiment reveals how many loops are vectorized by each compiler and which compiler outperforms the others in vectorization given a certain loop pattern. The other one quantitatively evaluates the evolution of a compiler by comparing code performance produced by two different versions of the compiler.

A. Comparison of vectorizers

Vectorization has grown in prominence with the wide deployment of vector devices in modern processors to improve single thread performance. In perfect situation, the speedup gained by vectorization could be close to the number of the vector lanes potentially used by the target data type. However, the overhead introduced during the vectorization process, e.g. pack/unpack operations, may neutralize or even negate the benefit in some circumstances. We define that a compiler effectively vectorizes a loop if $\frac{t_{scalar}}{min(t_{SSE},t_{AVX},t_{AVX2})} > 1.15$, where t_{scalar}, t_{SSE}, t_{AVX}, and t_{AVX2} are the execution time of a codelet when it is compiled with no vectorization, SSE, AVX, and AVX2, respectively. This condition specifies that vectorization is considered effective only when the speedup obtained from a vectorized codelet is greater than 1.15x.

Vectorized loop count: In order to fairly and accurately compare the vectorizers of the three compilers, we only count loops whose baseline execution time exceed 1000 cycles for every compiler. 959 loops are hence included in this analysis.

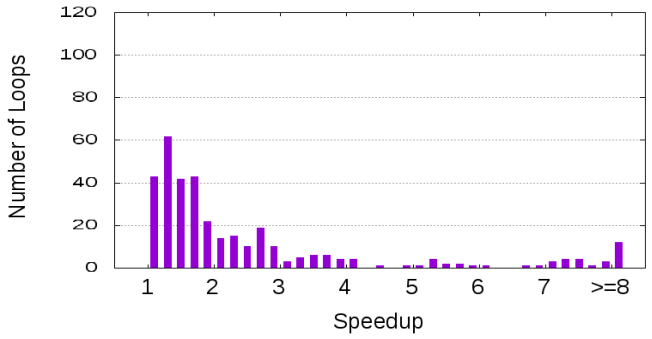
The second row in Table IV shows the number and percentage of loops that are vectorized effectively by default. The results indicate that ICC successfully vectorizes far more loops than GCC and Clang, and GCC outperforms Clang in vectorization by a relatively small margin. The difference in vectorization rate across compilers suggests that some compilers may find vectorization opportunities that the others do not. The reason that a compiler fails to automatically vectorize a vectorizable loop varies. Some typical causes include conservative dependence analysis, non-unit stride memory accesses, etc. These obstacles may be overcome by certain loop transformations, and then a compiler can proceed to vectorize a loop. We discovered from experiments that even a sequence of simple transformations can effectively make more loops vectorized. The third row in Table IV presents the number and percentage of additional loops that become vectorized after certain sequence of transformations are applied via our mutator. In this case, Clang benefits the most from the transformations; GCC comes second, and ICC receives the least benefit. Interestingly, the order is exactly opposite to that of the second row. This implies that a compiler tends to better benefit from source level transformations when its vectorizer is relatively naïve, and vice versa. The last row in Table IV lists the total number and percentage of loops that are vectorized after going through the mutation process.

	GCC 6.2.0	**ICC 17.0.1**	**Clang 4.0.0**
# of loops included	959		
# (%) of loops originally having effective vectorization	273 (28.5%)	395 (41.2%)	216 (22.5%)
# (%) of additional loops vectorized effectively and beneficial against the original baseline after mutation	77 (8.0%)	63 (6.6%)	98 (10.2%)
Subtotal	350 (36.5%)	458 (47.8%)	314 (32.7%)

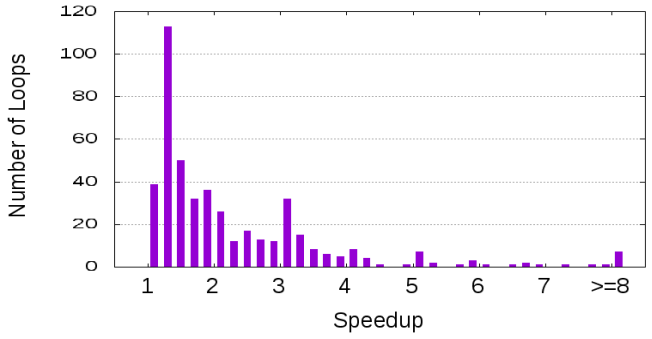
TABLE IV: Effective vectorization statistics

The results become more interesting as we further examine the distribution of speedup gained by vectorization from the target compilers (shown in Figure 2). Although the total number of loops effectively vectorized by GCC is notably less than that by ICC, GCC actually manages to help more loops obtain speedup above 4x. Meanwhile, both compilers have comparable results for speedup between 1.5x and 4x. But ICC significantly boosts more loops with 1.15x to 1.5x speedup. Therefore, GCC's vectorizer is on par with ICC's. Clang's vectorizer, on the other hand, rarely produces vector speedup that is higher than 4x. As a relatively young compiler,

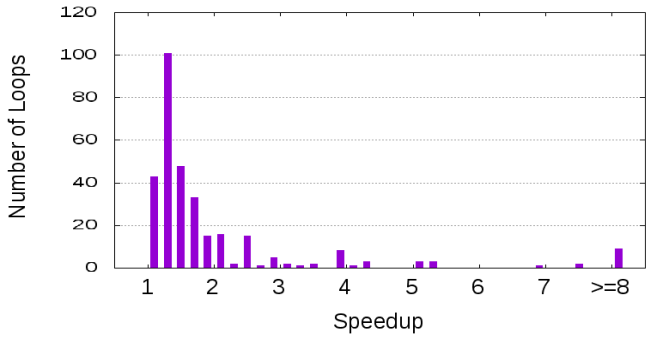
it seems that Clang has more room to improve its vectorization capability.



(a) GCC



(b) ICC



(c) Clang

Fig. 2: Distribution of vector speedup from effectively vectorized loops after mutations are applied

Effectiveness in common loop patterns: Thanks to the Clusterer, we are able to sort loops into categories and analyze them collectively. We studied a number of common loop patterns where one compiler's vectorizer outperforms the others. For illustrative purpose, two broadly used patterns are described, namely setting/zeroing a buffer and reduction.

Setting and zeroing buffers. Listing 1 contains a loop nest from the 183.equake application in SPEC2000. It stores a series of 1s into array `source_elms`. While both ICC and GCC can vectorize this loop nest easily using vector move instructions (e.g. `vmovdqu` or `vmovdqa`) and produce similar performance, the statically unknown variable `ARCHelems` in the loop conditional prevents Clang from optimizing this loop, leading to nearly 10x slowdown over either ICC or GCC.

As expected, Clang can vectorize the loop perfectly if `ARCHelems` is altered to an integer value (e.g. 1001) where the loop bound becomes known at compile time. But it fails to vectorize the code again if we make variable `i` global, e.g. defining it outside the loop scope. Clang's vectorization report prints that it cannot determine the number of loop iterations in this case. Therefore, GCC and ICC are likely to conduct more accurate analysis for code with this pattern to enable vectorization.

```
int i, ARCHelems, *source_elms;
for (i = 0; i <= ARCHelems - 1; i += 1) {
  source_elms[i] = 1;
}
```

Listing 1: Loop nest from 183.equake in SPEC2000 benchmark

Listing 2 is a codelet, extracted from the LU application in the NPB benchmark suite, that zeros two 2-D buffers. We observed that GCC was 1.97 and 1.87 times faster than Clang and ICC on this simple code, respectively. To understand this finding, we disassembled the binaries generated by these three compilers and learned how they generated code for the codelet.

- GCC called the `memset` function implemented in glibc to handle an entire row of a 2-D array at a time.
- ICC split the inner loop into two loops where one contained 64 iterations and the other had 2 iterations. It then vectorized the larger loop by a vector length of 4 and unrolled the vectorized loop 8 times, which was essentially 32 iterations in total. That is, this loop was executed only twice. In addition, ICC directly generated scalar code for the smaller loop since no loop optimization was considered profitable.
- While Clang also called the `memset` function to set these two arrays, it was more aggressive than GCC as it passed an entire 2-D array not a row to `memset` at once.

```
int i, k;
double phi1[66][66], phi2[66][66];
for (i = 0; i <= 64 + 1; i++) {
  for (k = 0; k <= 64 + 1; k++) {
    phi1[i][k] = 0.0;
    phi2[i][k] = 0.0;
  }
}
```

Listing 2: Loop nest froom LU in NPB benchmark

However, after changing 0.0 to 1.0 on the right hand side of the two statements, all three compilers vectorized the inner loop using AVX instructions, fully unrolled the loop (with a unroll factor of 8), and created an epilogue with 2 scalar move instructions for each statement. With this modification, all the target compilers produced similar performance effects.

We can thus safely come to the conclusion that the `memset` function in glibc probably vectorizes zeroing a buffer. There is a mutated version for ICC in our database that delivers the matching performance as GCC. By looking at the transformed code, we found that our mutator distributed the statements in the inner loop and then unrolled each distributed loop twice. As a result, ICC is not only capable of vectorizing the inner loop by a vector length of 4, but also able to fully unroll

the vectorized loop. Our mutator can also guide Clang to produce approximately the same performance by distributing the two statements in the inner loop. With this refinement, Clang can generate analogous assembly to the one emitted by GCC. Our repository, hence, is helpful in spotting I) the effectiveness of vectorization in different compilers and II) what transformations and in which order they can be applied to generate more efficient code.

Reduction. Listing 3 contains a representative reduction example. For this code, ICC has the best performance whereas GCC has the worst performance with Clang in between, i.e. ICC has 1.76x and 1.13x speedup over GCC and Clang, respectively. By inspecting the assembly code, we found that all three compilers vectorized this codelet, but GCC conservatively skipped unrolling and Clang aggressively unrolled the vectorized loop 20 times and ICC unrolled it with a unroll factor of 8.

This example indicates 1) the overhead of other loop optimizations, e.g. unrolling, might negate the benefit of vectorization when they are not applied cautiously, 2) the performance gain from unrolling varies significantly for different unroll factors. Hence, an accurate model for intelligent selection of proper transformations and their parameters is of significance. LORE is a tool that can be used to develop such accurate models.

```
int i;
float sum, a[32000];
for (i = 0; i < 32000; i++) {
  sum += a[i];
}
```

Listing 3: Loop S311 from TSVC benchmark

B. Comparison of GCC 6.2.0 against GCC 4.8.5

The second example analysis of our repository explores the performance difference of the loops over two compiler generations. We picked GCC 6.2.0 (later referred to as 6.2), released on August 2016, and GCC 4.8.5 (later called 4.8), released on June 2015, to measure the progress.

Effectiveness of mutations: Table V shows, on both compiler versions, the number of loops that are amenable to our mutator, the number of loops that are benefiting from our mutations (e.g. having at least 1.15x speedup), and the number of loops that are unfavorable to any possible mutations (e.g. having at least 1.15x slowdown). To minimize the affect of timing noise, we excluded loops with execution time less than 1000 cycles. As a result, only 1148 and 1118 out of 2499 loops remain in the results for GCC 4.8 and 6.2, respectively.

Table V exhibits that 46.3% and 43.2% loops in the repository are beneficial from our mutator for GCC 4.8 and 6.2, respectively. The percent of loops that are unfavorable to any mutations performed is 4.6% for GCC 4.8 and 5.5% for GCC 6.2. At least two possible reasons lead to the notable slowdown regardless whatever mutation is applied. First, the compilers can optimize the loop in the most appropriate way to achieve peak performance by themselves. Second, the number of transformations applied by our mutator is inadequate, i.e. other transformation techniques might be beneficial but not yet applied.

GCC	6.2.0	4.8.5
# of loops included	1118	1148
# (%) of loops having beneficial mutation(s)	483 (43.2%)	530 (46.3%)
# (%) of loops having all mutations unfavorable	61 (5.5%)	53 (4.6%)

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

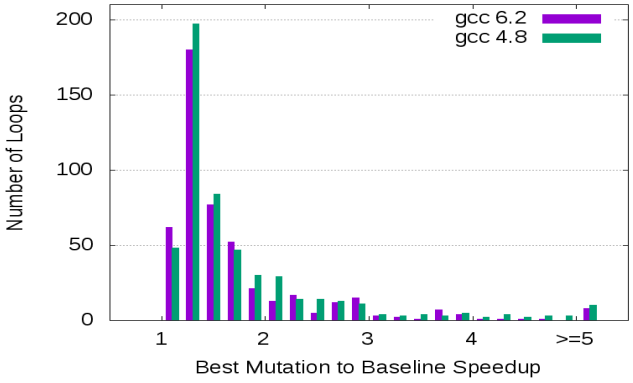


Fig. 3: Distribution of speedup from beneficial mutations for GCC 6.2.0 and 4.8.5

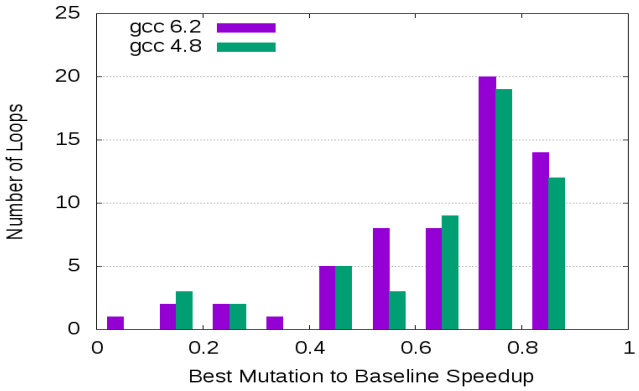


Fig. 4: Distribution of speedup from unfavorable mutations for GCC 6.2.0 and 4.8.5

Figure 3 and Figure 4 plot the distribution of speedup by the best mutation from loops containing beneficial mutation(s) or possessing only unfavorable mutations, respectively. The speedup values in the plots are calculated by $\frac{T_{baseline}}{min(T_{mutation})}$. Table V and the plots together reflect that mutations not only help more loops for GCC 4.8 than for 6.2, but also produce higher speedup for the former than for the latter. On the other hand, the number of loops that have only unfavorable mutations is higher with GCC 6.2 than with GCC 4.8, and the corresponding average slowdown is higher with the newer GCC version as well. These results imply that as GCC evolves, its optimization passes become more effective, which leaves less room for mutations to further optimize. Meanwhile, the improved optimizations can be disabled by mutations which

alter common loop patterns, and this may negatively impact performance, therefore causing higher numbers in Figure 4.

Performance improvement: We analyzed the performance discrepancy of loops compiled by GCC 4.8 and 6.2, respectively. This analysis only focuses on the 1113 loops that are shared by both versions. Table VI lists the numbers and percentages of loops receive either notable speedup or slowdown as GCC upgrades. The first column compares the baseline of the loops compiled by these two GCC versions. The speedup is calculated by $\frac{T_{4.8baseline}}{T_{6.2baseline}}$, and the slowdown is reciprocal. The second column compares the best mutations of a loop compiled by the two GCC versions. The speedup is calculated by $\frac{min(T_{4.8mutation})}{min(T_{6.2mutation})}$, and the slowdown is reciprocal. Note that here mutations also include the baseline, and the transformation sequences of the best mutations on the same loop for two GCC versions can be different. The speedups discussed above are also plotted in Figure 5. From the table and the plot, we can see that $20.0\% \sim 22.0\%$ of the loops run significantly faster with the more recent GCC. Although most of the speedup values are below 2x, some loops received performance boost up to 14x. By inspecting the assembly, we noticed that high speedups often came from loops that were not vectorized by GCC 4.8 but vectorized by GCC 6.2. On the other hand, $1.8\% \sim 3.1\%$ of the loops surprisingly executes much slower when compiled by the newer GCC version. In the worst case, the performance drop can be as severe as 4x. We observed that vectorization, or more specifically, vectorization profitability analysis also partially contributed to the performance loss.

From GCC 4.8.5 to 6.2.0	Baseline	Best mutation
# of loops shared	1113	
# (%) of loops having over 1.15x speedup	245 (22.0%)	223 (20.0%)
# (%) of loops having over 1.15x slowdown	34 (3.1%)	20 (1.8%)

TABLE VI: Number of loops that have notable performance boost or lose as GCC evolves from 4.8.5 to 6.2.0

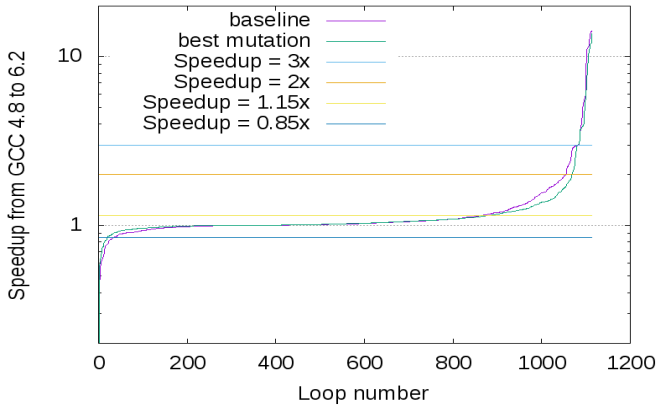


Fig. 5: Speedup from GCC 4.8.5 to 6.2.0 of all significant loops in the repository

The results also demonstrate that after applying various transformation sequences to a loop and selecting the best mutation, the performance disparity of the said mutation compiled by the two GCC versions tends to be less than the performance gap between the baseline. In Figure 5, it is clear that when speedup is greater than 1x, the line representing the performance gap between the best mutations is almost always under the line representing that between the baseline, and when speedup is less than 1x, the situation is completely opposite.

This observation indicates that some disadvantages of older optimization techniques can be mitigated by exploring the space of transformation sequences.

V. Related Work

While this is the first work that builds a systematic infrastructure to extract loops for optimization, analysis, and query, there are a few techniques proposed to implement the related functionality of some components in our framework. We classify them into 3 categories.

Repository. Fursin et al. [28] developed an open source knowledge management framework and repository, cTuning infrastructure, for automate performance tuning and optimization. It is able to do statistical analysis and modeling of empirical results for various benchmarks in their repository. cTuning offers compiler optimizations at the program level, e.g. using different algorithm parameters and compiler flags. However, our repository provides the community a large amount of loop nests with numerous mutations to achieve peak performance.

Extractors. To avoid tuning an entire large application, several previous papers [29], [30], [31] proposed to extract hot spots. Optimizations are then performed only on these small kernels to make the tuning process more manageable. The extractor in [29] outlined a code segment from a function and built a separate function for it. However, our extractor actually copies the whole target loop to a standalone file and creates declarations for all involved variables. The extracted codelet in our approach uses the sampled input data from the original program, which is compilable, executable, and measurable independently. Code Isolator [31] can statically analyze the data structures to be extracted but cannot handle pointers. On the other hand, our two-pass approach solves the pointer aliasing problem by capturing memory accessed by a loop nest and restoring the execution environment before launching the extracted loop.

Castro et al. [32] proposed a close approach, CERE, to extract and replay codelets from benchmarks for optimization. Although CERE also managed to extract hot loops and capture the machine states in the original program, our extractor differs from it in two aspects. First, CERE captures codelets at the LLVM Intermediate Representation (IR) level, thus losing the ability to explore source-level transformations. In contrast, our extractor is implemented at the source level. It not only preserves the exact behaviors of the original loop nest, but also makes the source-level optimizations viable. Second, CERE is tied to LLVM, which limits the portability of the codelets, but our extracted codelets can be directly fed to various compilers.

Mutators. Wolfe proposed Tiny [33] for loop restructuring using a number of loop transformation techniques. The main focus of Tiny was to check what transformations could be applied to a loop nest. However, the purpose of our mutator is to explore the search space and generate numerous valid mutations in the hope of enhancing performance. The LeTSeE project by Pouchet et al. [34] shares the same goal with our mutator. It explores the polyhedral transformation space of loops. The nature of polyhedral transformation makes it hard to derive the equivalent transformation sequence programmatically, therefore preventing us from using it to study the effect of transformation sequences on a more comprehensive set of loops in the repository.

VI. Conclusion and Future Work

This paper described a repository, LORE, which maintains a large amount of `C` language `for` loop nests extracted from benchmarks, libraries, and real-world applications. This repository, built by an array of tools, stores various information about each codelet, including execution time and performance counters. It not only provides abundant data to evaluate the pros and cons of a compiler on some optimizations, but also allows convenient measurement of the evolution of a compiler.

Two example experiments were described to demonstrate the value of the repository. One experiment measured the performance difference of auto-vectorizer from three production-quality compilers (ICC, GCC, and Clang). In this experiment, we found that compilers may be either too conservative or too aggressive in vectorization and other transformations, such as unrolling, for code with certain patterns. Our repository is able to provide more efficient transformations in some cases when commercial compilers fail. The other experiment gauged the evolution of GCC from version 4.8.5 to 6.2.0 where we found that the upgraded version sometimes emitted even less efficient code due to a number of reasons including lack of accurate profitability models and excessively aggressive instruction scheduling. We conclude that there is still a plenty of room to improve loop transformation techniques in the current compilers. We developed LORE as a tool that can be used by the community for this purpose.

The future directions of our research are 1) to integrate more loop transformation techniques, such as loop fusion, skewing, unswitching, etc., 2) to further improve the extractor to support loops with more complicated expressions, e.g. structure of arrays/structures, 3) to measure the importance (hotness) of a loop during extraction and record it in the database, 4) to evaluate the effectiveness of more compilers on more platforms, 5) to incorporate more applications in hot areas, e.g. big data, deep learning, etc. 6) to utilize machine learning to predict the best transformation sequence, compiler selection, and compiler flags for a given loop nest based on its features and the measurement data from the repository.

VII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Award 1533912.

References

[1] D. Cociorva, J. W. Wilkins, C. Lam, G. Baumgartner, J. Ramanujam, and P. Sadayappan, "Loop optimization for a class of memory-constrained computations," in *ICS*, 2001, pp. 103–113.
[2] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.
[3] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02–03, pp. 215–226, 2000.
[4] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, Mar. 1985.
[5] L.-N. Pouchet, "Polyopt/C: A polyhedral optimizer for the rose compiler," http://web.cse.ohio-state.edu/~pouchet/software/polyopt, 2011.
[6] "Xiph.Org Foundation," https://www.xiph.org.
[7] "The Lame Project," lame.sourceforge.net.
[8] "TwoLAME," www.twolame.org.
[9] "libdeep," github.com/bashrc/libdeep.
[10] "GAP," www.gap-system.org.
[11] "mozjpeg," github.com/mozilla/mozjpeg.
[12] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, "The ALPBench benchmark suite for complex multimedia applications," in *IISWC*, 2005, pp. 34–45.
[13] "ASC Benchmarks," https://asc.llnl.gov/sequoia/benchmarks/.
[14] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "Cortexsuite: A synthetic brain benchmark suite," in *IISWC*, 2014, pp. 76–79.
[15] "Fhourstones benchmarks," https://github.com/llvm-mirror/test-suite/tree/master/MultiSource/Benchmarks.
[16] P. Rundberg and F. Warg, "The FreeBench v1. 0 benchmark suite," *URL: http://www. freebench. org*, 2002.
[17] R. F. Van der Wijngaart and T. G. Mattson, "The parallel research kernels." in *HPEC*, 2014, pp. 1–6.
[18] "Livermoore loops," http://www.roylongbottom.org.uk/livermore%20loops%20results.htm.
[19] "Media Bench II," http://mathstat.slu.edu/~fritts/mediabench.
[20] S. Browne, J. Dongarra, E. Grosse, and T. Rowan, "The Netlib mathematical software repository," *D-Lib Magazine, Sep*, 1995.
[21] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The NAS parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.
[22] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.
[23] "SciMark 2.0," http://math.nist.gov/scimark2.
[24] "SPEC benchmarks," http://www.spec.org.
[25] "Extended test suite for vectorizing compilers," http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz.
[26] "Intel® 64 and IA-32 Architectures Software Developer's Manual," http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf.
[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005, pp. 190–200.
[28] G. Fursin, "Collective knowledge," http://github.com/ctuning/ck/wiki.
[29] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas, "Effective source-to-source outlining to support whole program empirical optimization," in *LCPC*, 2010, pp. 308–322.
[30] C. Akel, Y. Kashnikov, P. d. O. Castro, and W. Jalby, "Is source-code isolation viable for performance characterization?" in *ICPP*, 2013, pp. 977–984.
[31] Y. Lee and M. Hall., "A code isolator: Isolating code fragments from large programs," in *LCPC*, 2005, pp. 164–178.
[32] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby, "Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization," *ACM TACO*, vol. 12, pp. 6:1–6:24, 2015.
[33] M. Wolfe, "Experiences with data dependence abstractions," in *ICS*, 1991, pp. 321–329.
[34] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part ii, multidimensional time," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 90–100.