

MCompiler: A Synergistic Compilation Framework

Aniket Shivam

*Department of Computer Science
University of California, Irvine
Irvine, CA, USA
aniketsh@uci.edu*

Alexandru Nicolau

*Department of Computer Science
University of California, Irvine
Irvine, CA, USA
nicolau@ics.uci.edu*

Alexander V. Veidenbaum

*Department of Computer Science
University of California, Irvine
Irvine, CA, USA
alexv@ics.uci.edu*

Abstract—This paper presents a meta-compilation framework, the *MCompiler*. The main idea is that different segments of a program can be compiled with different compilers/optimizers and combined into a single executable. The *MCompiler* can be used in a number of ways. It can generate an executable with higher performance than any individual compiler, because each compiler uses a specific, ordered set of optimization techniques and different profitability models and can, therefore, generate code significantly different from other compilers. Alternatively, the *MCompiler* can be used by researchers and compiler developers to evaluate their compiler implementation and compare it to results from other available compilers/optimizers.

A code segment in this work is a loop nest, but other choices are possible. This work also investigates the use of Machine Learning to learn inherent characteristics of loop nests and then predict during compilation the most suited code optimizer for each loop nest in an application. This reduces the need for profiling applications as well as the compilation time.

The results show that our framework improves the overall performance for applications over state-of-the-art compilers by a geometric mean of 1.96x for auto-vectorized code and 2.62x for auto-parallelized code. Parallel applications with OpenMP directives are also improved by the *MCompiler*, with a geometric mean performance improvement of 1.04x (up to 1.74x). The use of Machine Learning prediction achieves performance very close to the profiling-based search for choosing the most suited code optimizer: within 4% for auto-vectorized code and within 8% for auto-parallelized code. Finally, the *MCompiler* can be expanded to collect metrics other than performance to be used in optimization process. The example presented is collecting energy data.

Index Terms—Compiler Optimizations, Loop Transformations, Machine Learning, Compilation Framework

I. INTRODUCTION

An important compiler task is optimizing applications for better performance on a target architecture. Optimizing loop nests, in particular, contributes significantly towards achieving better performance. State-of-the-art architectures have multiple cores on a chip, where each core has Single Instruction Multiple Data (SIMD), or vector, capabilities. These architectural features provide opportunities for a compiler to expose parallelism in applications on multiple levels. The code optimization techniques to *auto-vectorize* the loop nests [1], [32], [44], so as to generate SIMD instructions, require careful analysis of data dependences, memory access patterns, etc. Several auto-parallelization techniques [8], [14], [22]–[25], [31] and directive based parallel programming models, such as OpenMP [30], have been developed to take advantage of

multiple cores. In fact, most auto-parallelization implementations in modern compilers, which take serial code as input, generate OpenMP code.

Key loop transformation techniques [5], [21], [44] include Distribution, Fusion, Interchange, Skewing, Tiling and Unrolling. Code optimizers search for an optimal semantic-preserving sequence of transformations to generate a better performing code, either serial or parallel. But evaluating if a sequence of transformations is optimal is complex and the search for the best sequence of transformations and their profitability is guided by heuristics and/or approximate analytical models. Thus, a code optimizer may end up with a sub-optimal result and different code optimizers may, for the same source code segment, generate code with significant performance differences on the same architecture. Some major challenges in developing the heuristics and profitability models is predicting the behavior of a multi-core processor which has complex pipelines, multiple functional units, complex memory hierarchy, hardware data prefetching, etc. Parallelization of loop nests involves further challenges for the code optimizers, since communication costs based on the temporal and spatial data locality among iterations have an impact on the overall performance. Evaluation studies [17], [26], [28], [40] have shown that state-of-the-art code optimizers may miss out on opportunities to auto-vectorize and auto-parallelize the loop nests for modern architectures. For optimizing applications written in C, there are several compilers and domain specific loop optimizers that perform auto-vectorization and, in some cases, auto-parallelization of code. From a given code optimizer’s point of view, the sequence it used is the best but there is no way of knowing how close it gets to optimal performance or if there is any headroom for improvement.

This paper presents a compiler framework, *MCompiler*, that allows each loop nest to be optimized by the best optimizer available to it. The *MCompiler* incorporates code optimizers from Intel’s C compiler, PGI’s C compiler, GNU GCC, LLVM Clang. In addition to these, two Polyhedral Model based loop optimizers, Polly [18], [34] and Pluto [9], [33] are used. The *MCompiler* identifies loop nests from C applications, optimizes the loop nests using different code optimizers, profiles each optimized code version as part of the applications, and links the best performing codes to generate the complete application binary. The best loop nest code selection allows the *MCompiler* to produce higher-performing code than the best

of the compilers in the framework. The framework allows for easy integration of newer versions and newer configurations of the available code optimizers and also allows the addition of new code optimizers.

The framework can be used to optimize applications, first, for serial execution with auto-vectorization of loop nests. This optimizes loop nests for SIMD or vector code generation, in addition to optimizing loop nests for data locality, memory hierarchy, etc. Second, our framework can also target multi-core processors, by taking serial loop nest codes as input and auto-parallelizing those loop nests using the available code optimizers to generate multi-threaded code. Auto-parallelized code is also optimized for SIMD execution over each thread. In this case, the original loop nests are transformed such that loop iterations can be reordered and scheduled for parallel execution across the multiple cores. Third, our framework can target OpenMP applications, i.e., applications with OpenMP directives inserted across sections of the code meant for parallel execution. The performance evaluation of the *MCompiler* shows that our framework indeed achieves an overall geometric mean speedup of 1.96x for serial code, 2.62x for auto-parallelized code and 1.036x (up to 1.74x) for OpenMP code over Intel C Compiler for applications from various benchmark suites.

The framework extracts loop nests from the applications' source files into separate source files as a function, together with any additional information needed. It then replaces loop nests with a function call in the original source files. This allows for separate code optimizers to focus on just the loop nests and also allows the framework to insert the best performing code, i.e., linking object files to generate the executable. To evaluate the speedup potential, the framework initially optimizes each extracted loop nest with all available code optimization *candidates*. The performance of each optimized loop nest is measured as part of the complete application and allows for selecting the best performing code for each loop nest. This step is highly time consuming, it was used to establish that the framework can indeed improve performance. The final step links the selected object files generating the executable for the complete application.

The second goal of this paper is to use Machine Learning (ML) models to predict the most suited code optimizer for a given loop nest. This can eliminate the profiling in the framework and should reduce compilation time. However, this can lead to a potential performance loss compared to profiling due prediction errors. The Machine Learning model or *classifier* can predict a different optimizer than the profiling-based best code optimizer.

The input or *features* to our Machine Learning model are hardware performance counters collected from profiling a serial (-O1) version of a loop nest. Embedding Machine Learning models in compilers is continuously being explored by the research community [3], [11], [12], [16], [27], [37]–[40], [42], [43]. Most of the previous work used Machine Learning in the domain of auto-vectorization, phase-ordering and parallelism runtime settings. This work applies Machine

Learning on a coarser level, in order to predict the most suited code optimizer - for serial as well as parallel code.

Previous studies have shown that hardware performance counters can successfully capture the characteristic behavior of a loop nest. Machine Learning models in those studies either used a mix of static features (collected from source code at compile time) and dynamic features (collected from profiling) [40], [42], or exclusively use dynamic features [3], [12], [37], [43]. The approach used in this paper belongs to the latter class and exclusively uses hardware performance counters collected for a loop nest. The framework makes the predictions for the most suited code optimizer using the incorporated, trained ML model as it compiles each loop nest. This replaces the expensive profiling for every optimizer with a single profile of each nest to collect hardware performance counters. They are then used as features to make the prediction from the trained ML classifier. The functionality to predict the most suited code optimizer for loop nests can also be embedded in a Just-In-Time (JIT) compiler, but this is part of future work. It would use run-time profiling, for instance of a subset of iterations, to collect hardware performance counters and then use them to make predictions.

The evaluation of the *MCompiler* with Machine Learning predictions shows that the performance of applications is within 4% for auto-vectorized code and within 8% for auto-parallelized code compared to the profiling-based search for the most suited code optimizer. We exclude OpenMP applications from ML predictions because in this case it is the user-inserted directives that make most of the difference to the performance rather than the inherent characteristics of loop nests. Hence, this problem is not suitable for such predictions.

The paper also highlights the usability of the *MCompiler* framework for compiler researchers to evaluate their optimization techniques or improvements against other available code optimizers and compilers. To compile an application using the *MCompiler* framework, the user needs a few modifications to the build configurations, similar to what is required to add any other compiler. But no modifications to the source code of the applications are required.

The paper also shows how the *MCompiler* framework can be extended with additional metrics to allow users to gain insight into energy consumption on target architecture. This highlights the potential for the expansion of the *MCompiler* framework.

Overall, this paper makes the following contributions:

- It presents a meta-compilation framework that improves performance for C applications for serial as well as parallel execution, including OpenMP applications.
- It demonstrates that prediction for the most suited code optimizer (serial as well as parallel) for a loop nest can be accurately made using Machine Learning classifiers.
- It presents the *MCompiler* framework extension for reporting energy consumption per loop nest.
- The framework will be open sourced for researchers and compiler developers to analyze and compare their code optimization techniques.

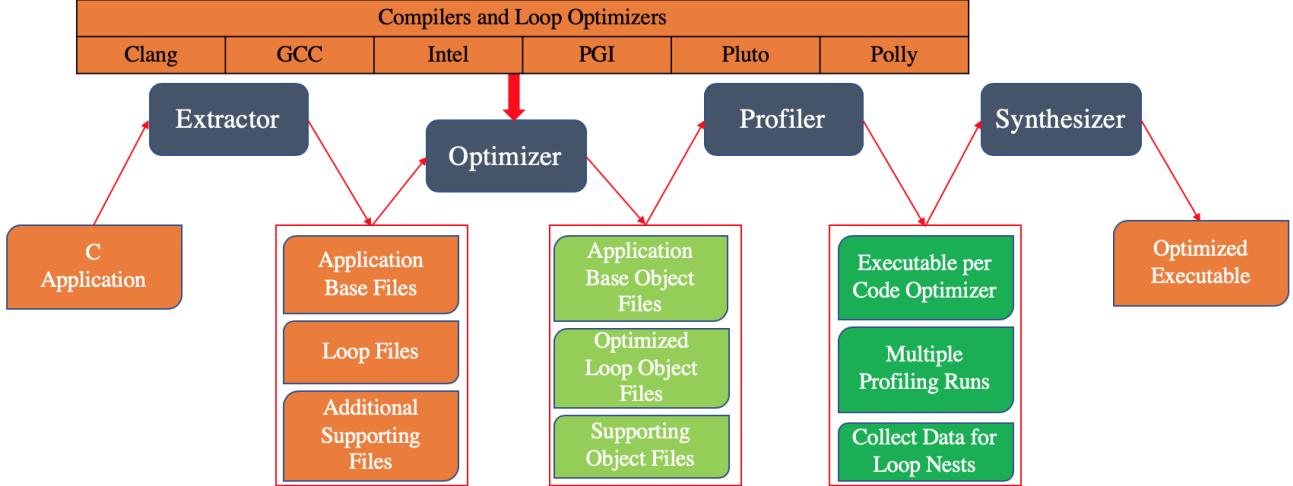


Fig. 1: *MCompiler* Framework

The rest of the paper is organized as follows. Section II describes the *MCompiler* framework and the methodology for choosing the most suited code optimizer for a loop nest using profiling-based search as well as using ML-based prediction. Section III describes the evaluation methodology and analyzes the experimental results. Section IV discusses related work. We conclude the paper with Section V.

II. FRAMEWORK DESIGN AND IMPLEMENTATION

This section describes the overall architecture of the *MCompiler* framework and the technical details about the individual phases of the framework. The specifics of the architecture for incorporating Machine Learning predictions is discussed later in the section.

A. Overall Framework Architecture

Figure 1 shows the structure of the *MCompiler* framework. The first phase is the *Loop Extraction* from C applications. Since loop nests dominate the performance in various applications, the *Extractor* parses the source files to find `for` loop nests, extract those loop nests as functions into separate independently compilable files and replace the loop nests with the corresponding function call in the *base* source file. Base files are similar to the original source files but with loop nests replaced with function calls. Whereas loop files are newly generated files which define the function containing the loop body and supporting components to make them compile successfully. This Extractor is inspired by the loop extractor described in the work by Chen et. al. [13] to encapsulate loop nests into standalone executables.

The second phase is the *Optimization* phase. The *Optimizer* compiles each loop file with the available code optimizers. Also, it compiles the base files and additional *MCompiler* files, i.e., files added to support the functioning of the framework. For source-to-source code optimizers, a *default compiler* is used to compile optimized loop files, the base files and additional files.

The third phase is the *Profile* phase, where the application is profiled for execution times of the extracted loop nests. Executables generated for each code optimizer is executed and reported execution times for the loop nests are collected.

The final phase is the *Synthesis* phase. Here, for each extracted loop nest, the collected loop execution times are compared from every code optimizer and a code optimizer that produced the best performing code is selected, i.e., the optimized code that completes the execution of the loop body in the shortest time. Finally, the default compiler is used to link the object files from the selected code optimizer for every loop file, plus the object files generated by the default compiler for the base files. This step also requires linking libraries that code optimizers may have used or taken support of for generating code for the loop files.

For large applications, if `-c` flag is provided, i.e., compile to object files only, then just the Extractor and the Optimizer are enabled. In such cases, the Profiler and the Synthesizer are enabled only at link-time. *MCompiler* framework handles flags for macro definitions, paths to header files and libraries for linking, etc. similar to other compiler.

B. Loop Extraction Phase

The loop extractor works in three phases and is implemented using ROSE, a source-to-source compiler infrastructure [36]. First, the extractor traverses the abstract syntax tree (AST) and locates the `for` loop nests that are eligible for extraction. Second, the extractor creates a new file for this loop, adds necessary headers and macro definitions in the loop file, and also add `extern` declaration for global variables and global functions, as well as, for functions called in the scope of the loop body. Encloses the loop body in a function definition with parameters being the variables and pointers to the data structures required by the loop body in order to compile and run correctly. Third, in the base file's AST replace the loop body with a function call (with required arguments) and add

Code Optimizer	Version	Flags (Auto-Parallelization flags)	Auto-Parallelization
clang (LLVM)	7.0.0	-Ofast -march=native	No
gcc (GNU)	5.4.0	-Ofast -march=native	No
icc (Intel)	18.0.0	-Ofast -xHost (-parallel)	Yes
pgcc (PGI)	18.10	-fast -tp=skylake -Mllvm (-Mconcur)	Yes
PLuTo (source-to-source) + icc	0.11.4	--tile (--parallel)	Yes
polly	7.0.0	-O3 -march=native -polly -polly-tiling -polly-vectorizer=stripmine (-polly-parallel)	Yes

TABLE I: Candidate Code Optimizer and their flags

an `extern` declaration to this function. Finally, generate the modified base source file and the new loop files.

While traversing the AST for eligible loop nests, the extractor skips loop nests with irregular control flow that hinders extraction, i.e., contains `return` and `goto` statements. Also, it skips loop nests with calls to static functions and static variables since those properties hinder their usage in the new loop files.

The extractor generates two similar versions of the loop files, where one version contains extra code around the loop body to collect profile information about the loop nests. The version with the profile code is used during the Profile phase and is responsible for generating information regarding the execution time of the loop nests. The other does not contain any profile code and is used while generating the final executable for the applications.

1) *Function Definition enclosing the Loop Nests:* The extractor generates the lists of variables, with their data types, used inside the scope of the loop body. All primitive data types (`int`, `float`, etc.) are passed by reference, as well as the user-defined types such as arrays, structs and `typedefs`. The extractor also does an optimization to maintain properties of the loop from the point of view of the code optimizers. This optimization copy the function parameters of primitive types (passed by reference) into local variables (with same names as original variables) before the loop body and correspondingly copy the local variables into the function parameters at the end of the loop body. This optimization prevents any change to loop body and is also critical to performance since usage of pointers can prevent some code optimizations.

For loop nests with OpenMP directives, the extractor moves the directives with loop body and sanitizes the clauses of variables that are not present in the scope of the loop nest. For OpenMP `for` loops that are enclosed in a `omp parallel` region, extracting the loop body with `omp for` directive doesn't change the behavior of the program. One drawback of extracting OpenMP `for` loops that are enclosed in a `parallel` region in such manner is that in the presence of `threadprivate` variables, synthesizer encounters a link-time error because compilers may generate different symbols for the same `threadprivate` variable.

C. Optimization Phase

The framework uses six candidate code optimizers: Intel's `icc`, PGI's `pgcc`, GNU's `gcc`, LLVM `clang`, LLVM based

polyhedral loop optimizer `Polly` and source-to-source polyhedral loop optimizer `Pluto`. We chose `icc` as the default compiler because its performance is, on average, the best of the compilers included. It is also used to compile source files generated by source-to-source loop optimizer, i.e., Pluto. Table I shows the flags used for optimizing loop nests for serial execution and parallel execution. These flags also include target architecture specific flags to enable optimizations that can generate better performing code on the specific architecture. For OpenMP applications, flags from serial configuration are used in addition to the OpenMP flags.

The optimizer can compile loop files and base files in parallel. This is similar to `-j` option of Makefiles, but here all candidate code optimizers are invoked in parallel to compile the source files. This reduces the overall compilation time for the *MCompiler* framework. The optimizer generates multiple executables of the application (with profile code) where each executable is completely compiled and linked by a candidate code optimizer.

D. Profile Phase

The profiler executes the executables generated by the code optimizers one-by-one and performs multiple runs for stable data, if requested. Profiler at the end of each execution collects the profiled information for each of the loop nests and forwards it to the Synthesizer. For applications that need input through command line, the profiler runs the application with the input given to the *MCompiler* framework using a `--input` flag.

E. Synthesis Phase

The synthesizer compares the collected profile information, i.e., the execution times for each loop nests from different code optimizers and chooses the code optimizer that performed the best as the most suited code optimizer. For loop nests with no profile information, i.e., the code that was not executed during profiling, the default compiler is chosen as the most suited code optimizer. The synthesizer then generates the final executable that contains no profile code. For OpenMP application, the synthesizer links OpenMP runtime libraries that are used by different compilers, e.g., `icc` and `clang` use compatible OpenMP runtime libraries whereas `gcc` doesn't. Static libraries specific to compilers are also linked to successfully generate the final executable.

F. Framework Architecture for Machine Learning Predictions

The framework for choosing the most suited code optimizer for the loop nests using the ML predictions is shown in figure

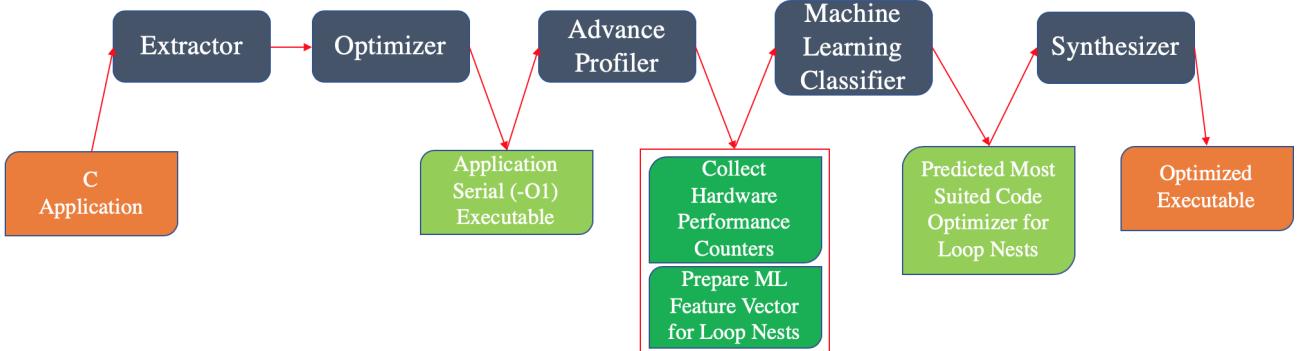


Fig. 2: *MCompiler* Framework with Machine Learning Predictions

2. The ML predictions are used to predict the most suited code optimizer for both serial, auto-vectorized code as well as auto-parallelized code. The input to the ML Classifier for making the predictions are the hardware performance counters for the loop nests. This strategy for collecting hardware performance counters for the loop nests and using them to predict the most suited code optimizer is inspired from the work of Shivam et. al. [37]. The architecture of the *MCompiler* framework is modified in the following ways to predict the most suited code optimizer for the loop nests.

First, the Optimizer now generates an additional executable that is compiled by the default compiler for serial execution with -O1 optimization level. Second, the Profiler is replaced by the Advance Profiler for making ML predictions. Advance Profiler executes the serial (-O1) execution and collect hardware performance counters for the loop nest. If the loop nest is not executed or the hardware performance counters are not present (happens for loop nests with very few computations), the default compiler is chosen by the Synthesizer.

Next, the collected hardware performance counters for each loop nest are transformed into the feature vector, i.e., the input to the ML classifier. Third, the ML classifier makes the prediction for the most suited code optimizer for a loop nest based on the feature vector. The ML classifier is a trained ML model. There are two separately trained ML models, one for serial code predictions while the other is for parallel code predictions. Finally, these predictions from the ML classifier are forwarded to the Synthesizer, which uses the code optimizer from the prediction to link the correct optimized loop object files and generate the final executable for the application.

The *MCompiler* driver invokes the ML prediction part of the framework over the original *MCompiler* flow with profiling-based search if the --predict flag is provided. The ML models are trained and incorporated in the *MCompiler* framework using OpenCV's Machine Learning module [29].

1) Collecting Hardware Performance Counters for the Loop Nests: The features, i.e., the hardware performance counters used for the Machine Learning models are collected by profiling loop nests using Intel's VTune Amplifier. We use generated code from Intel compiler to generate the executable that is

then used for profiling. All the loop optimizations are disabled during this compilation by using the -O1 flag. In addition to that, the optimization that are responsible for vector code generation and parallel code generation are disabled too. The profiling information, therefore, provides an insight into the characteristics of the loop nests while eliminating the influence of compiler transformations and behavioral changes incurred from special architectural features of the underlying architecture. The performance counters that are collected include, but not limited to, instruction-based (instruction types and counts) counters, CPU clock cycles-based (including stalls) counters, memory-based (D-TLB, L1 cache, L2 cache, L3 cache) counters.

Once the hardware performance counters are collected for the loop nests, we skip dynamic instruction count as a feature and normalize the rest of the hardware performance counters in terms of *per kilo instructions* (PKI). Based on our analysis, this allows the Machine Learning models to learn about the inherent characteristics of the loop nests and not bias them towards characteristics such as loop trip count.

2) Random Decision Forest Classifier (RF): We chose Random Decision Forest [19] as our classification algorithm to predict the most suited code optimizer for the loop nests. RF is a learning algorithm that builds on the principles behind Decision Trees. Generally, the Decision Tree algorithm, learns from training data by building a structured and hierarchical representation of the correlation between features and classes. Features represent the nodes in the trees and classes are leaves at the deepest level. An optimal Decision Tree would perfectly and accurately divide the data among the target classes. However, finding an optimal tree is an NP-Complete problem, therefore we have to rely on heuristics such as greedy search.

Decision Trees suffer from several issues. The main one being a tendency towards overfitting, that is, the tree loses generalization the deeper the tree goes, modeling the trend for training data but be inaccurate for new instances. RF provides a better solution for overfitting and classification bias by adding two stochastic steps to the Decision Tree Algorithm. From the training dataset, RF creates a bootstrapped subset by stochastically choosing the instances or features (with repeats

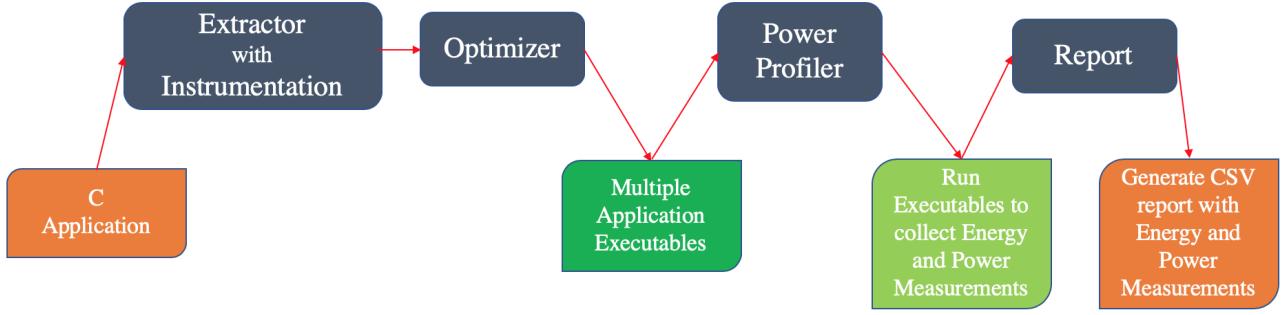


Fig. 3: *MCompiler* Framework for collecting Energy and Power Measurements

allowed) that will be used for building the decision trees. This is called Bootstrap Aggregating and the instances inside the created subset are called In-Bag instances. After creating the Bagged dataset, an arbitrary number of decision trees are built using subsets of randomly chosen features. The depth of the trees is limited by the number of features allowed (according to a predetermined threshold) and by another arbitrary number. Each decision tree accuracy is evaluated using the remaining (out-of-bag) instances that weren't part of the Decision Tree building phase.

Classification is achieved through a voting algorithm, where a target value is generated by each Random Tree, the one with the highest number of trees will be the class assigned to the new instance.

Since Random Forest is constantly evaluating the performance of the subsets of features, we can easily detect the ones that were used in the better performing trees. Therefore, we require little to no feature filtering before running the algorithm. This is useful when approaching a new problem where the correlation between the input features and the output class is not entirely known.

G. Tool for Compiler Researchers

The *MCompiler* framework, we believe, is also an important tool for compiler researchers who regularly implement and test their optimization techniques and/or tweak analytical or heuristic models for improving performance for applications. The framework design also allows for adding new code optimizers and monitoring their performance on entire application or just on particular hotspots. The command line options for the *MCompiler* framework are shown in figure 4.

The framework also allows for training new Machine Learning models and using them for making predictions. Various flags are available for choosing the target architecture and choosing particular optimizations such as auto-parallelization optimizations or enabling particular passes such as data prefetching pass. The framework also allows for running Advanced Profiler independently, i.e., collect hardware performance counters for all the hotspots in an application while disabling the ML predictions.

```

$ MCompiler
Usage: MCompiler <input_files> [options] [-o output]
Options:
  -h,--help          Print usage
  --[no]extract      Extract hotspots
  --[no]profile      Profile extracted hotspots
  --[no]synthesize   Combine best performing hotspots to generate binary
  --adv-profile     Advanced Profiling
  --power-profile   Power Profiling
  --predict         Predict candidate using ML
  --test            Test performance compared to other compilers optimized code
  --parallel        Generate multi-threaded code based on OpenMP directives
  Default: Serial code generation (with vectorization)
  --auto-parallel   Auto-parallelize the hotspots
  --extractkernel   Extract consecutive loop nests, if possible.
  --prefetch        Enable software data prefetching
  --profile-runs=<num> Number of time profiler should run the program to collect data.
  --input=<args>    Input to the program Needed to generate profiling information
  --predict-model=<args> Path to the trained ML model.
  --haswell          Compile for Intel Haswell processor. Default: Skylake
  --knl             Compile for Intel Knights Landing processor. Default: Skylake
  --skylake         Compile for Intel Skylake processor.
  --c99             Conforms to ISO C99 standards. Default: C11
  -j                Compile hotspots in parallel
  -c[<arg>]         Compile to object file
  -o[<arg>]         Output object/binary name
  -l[<arg>]         Directory to include file search path
  -L[<arg>]         Directory to search for libraries
  -l[<arg>]         Instruct the linker to link in the -l<string> library
  -D[<arg>]         Macro definition
  --debug           Output MCompiler workflow
  --info            Print information for MCompiler workflow
  --novec          Disable vectorizer
  --disable-polyhedral Disable Polyhedral Model based loop optimizers

```

Fig. 4: *MCompiler* command line options

H. Instrumenting Loop Nests to Measure Energy Consumption

The framework is designed to be extensible and add other features, in addition to choosing or predicting the most suited code optimizer for loop nests as based on performance. One such feature very much sought out by developers today is energy consumption analysis and optimization. This sub-section describes the addition of the energy measurement option for a loop nest. The *MCompiler* driver invokes this part of the framework, as shown in figure 3, if the `--power-profile` flag is set. The modifications to the framework required for collecting and reporting energy measurements are as follows.

First, the Extractor instruments the loop nest body with LIKWID [41] APIs. LIKWID uses the RAPL interface [20, Chapter 14.9] to measure the consumed energy on the package (socket) and DRAM level. The Extractor adds `LIKWID_MARKER_INIT` and `LIKWID_MARKER_START(<LOOP ID>)`

statements before the loop nest body and adds LIKWID_MARKER_STOP(<LOOP ID>) and LIKWID_MARKER_CLOSE statements after the loop nest body. Second, the Optimizer compiles the loop files with an additional macro definition -DLIKWID_PERFMON. Third, the Power Profiler generates an executable, each optimized by one of the six code optimizers as mentioned in Table I. Next, the Power Profiler runs the executables with likwid-perfctr which pins the application to a particular processor and produces the energy measurements. Finally, the Power Profiler generates the CSV report with energy and power results for each loop nest in the applications.

The goal of adding this feature to the *MCompiler* framework is to provide a user with more information and insight about the application. Furthermore, this feature can be used to generate code that minimizes energy consumption on intended architectures and not just the execution time. Or to optimize the energy-delay product.

I. Expanding the Framework with more Code Optimizers and/or with Optimizer Flag Combinations

The *MCompiler* framework allows for addition of more code optimizers so as to give more options for generating the optimized applications. In addition to that, the framework allows for adding different combinations of compiler flags or code optimizer flags to optimize the applications. This allows users to explore how different code optimizer flags impact the performance of the applications and use *MCompiler* framework to generate even better performing executables. By its design the framework can also include auto-tuning frameworks, such as domain-specific auto-tuner called Open-Tuner [2], for optimizing applications. Exploring different combinations of code optimizer flags is beyond the scope of this work. In this work, we present results with applications optimized using the most influential or recommended flags combinations, for improving performance, from each code optimizer.

III. EXPERIMENTAL ANALYSIS

This section describes the experimental methodology and present the results and their analysis.

A. Benchmarks, Code Optimizers and Target Architecture

Several different benchmark suites are used to evaluate the effectiveness of the *MCompiler* framework. One benchmark suite used is Test Suite for Vectorizing Compilers (TSVC) by Callahan et al. [10] and Maleki et al. [26]. This benchmark was developed to assess the auto-vectorization capabilities of compilers. Therefore, these loop nests are only used in the serial code related experiments. The second benchmark suite used is Polybench [35]. This suite consists of 30 benchmarks that perform numerical computations used in various domains, such as linear algebra computations, image processing, physics simulation, etc. The benchmarks in Polybench have been demonstrated to have performance gain on parallelization, therefore these loop nests are used for auto-parallelized code

experiments as well. The third benchmark suite is NAS Benchmark Suite [4], especially, NPB3.3-SER, NPB3.3-OMP and NPB-ACC [45]. These benchmarks are used in serial code, auto-parallelized code and OpenMP parallel code experiments. Lastly, a set of C benchmarks from SPEC OMP 2012 was used for OpenMP experiments. The train dataset was used for profiling SPEC benchmarks, whereas the results are shown for ref dataset. Table I showed the six code optimizers that have been incorporated in the *MCompiler* framework. All six optimizers are used for serial and OpenMP experiments. Of the six optimizers, only four optimizers (icc, pgcc, Polly and Pluto) can auto-parallelize the serial code and are used for auto-parallelized code experiments. The baseline for performance comparison is icc (-Ofast -xHost [-parallel]) compiled benchmarks for all experiments. icc was chosen as the baseline because icc generated code performed better for more benchmarks than other code optimizers. The source codes used for the baseline are the original benchmark codes and not the modified source codes generated by the *MCompiler*'s Loop Extractor.

The target architecture for our experiments is a two-socket, sixteen-core Intel Skylake Xeon Gold 6142. Each Xeon processor has 32KB L1 cache, 1MB L2 cache, 22MB L3 cache. The Skylake architecture supports SIMD instruction set extensions, i.e., SSE, AVX, AVX2, AVX-512CD and AVX-512F. Turbo boost is switched off, cores are operating at the maximum frequency, i.e., 2.6 GHz. For the auto-parallelization and OpenMP experiments, only one thread is mapped per core by setting the environment variables for OpenMP runtimes.

B. MCompiler Profiling-Based Search

This section presents experimental results using the exploratory search by the *MCompiler* for choosing the most suited code optimizer. Each application was profiled 3 times for each of the code optimizers and the median execution time was chosen for deciding the most suited code optimizer.

1) *Serial Code*: The results are shown in Figure 5 with benchmark labels showing the dataset set size in parenthesis and the benchmark suite that a particular benchmark belongs to. The GeoMean speedup across the 151 loop nests from TSVC is 1.34x over icc. The performance of *MCompiler* for Polybench benchmarks is usually better than or equal to icc while considering the overheads from *MCompiler* loop extractions. As expected, the two polyhedral model based optimizers were chosen as the most suited code optimizer for 55% of the loop nests from Polybench. Whereas, for 158 out of 306 (51%) loop nests from NPB benchmarks, icc is chosen as the most suited code optimizer.

2) *Auto-Parallelized Code*: These experiments were performed with 32 threads for both profiling the applications and evaluating the performance. The code optimizers optimized the loop nests with their default setting for statically deciding the profitability of the parallel code and for choosing the runtime settings, such as scheduling policies.

Benchmarks from Polybench, NPB-OMP and NPB-ACC were used in these experiments. Polybench was shown to have

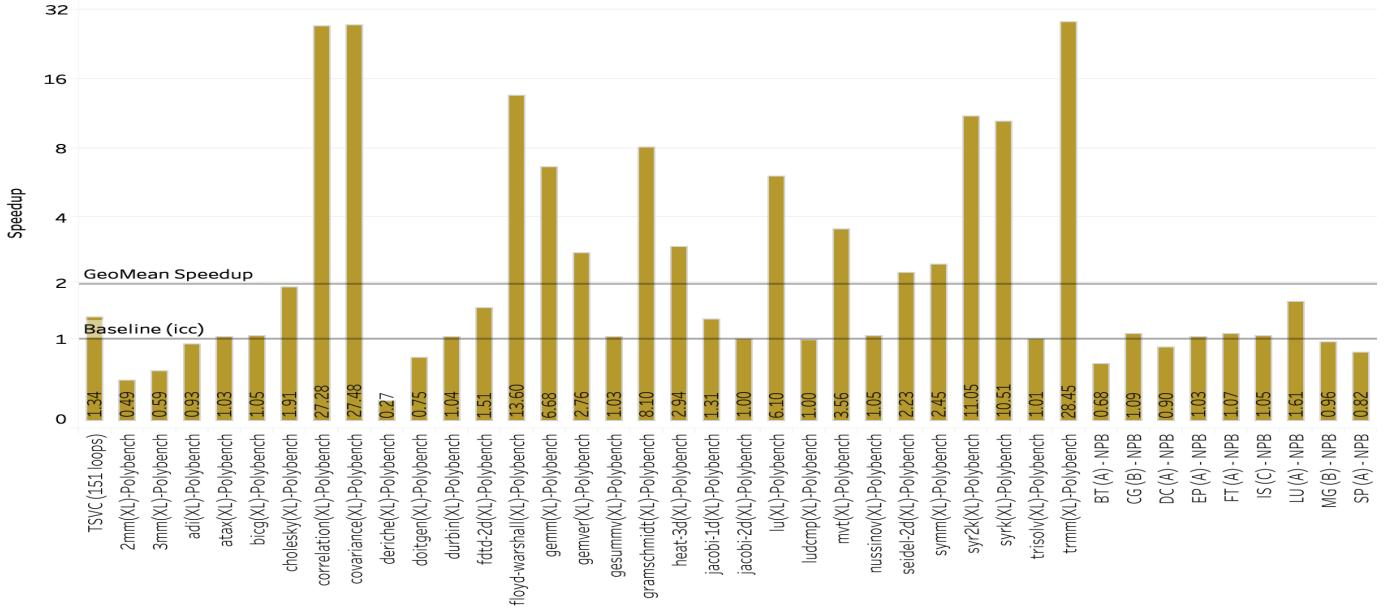


Fig. 5: *MCompiler* Speedup for Serial Benchmarks

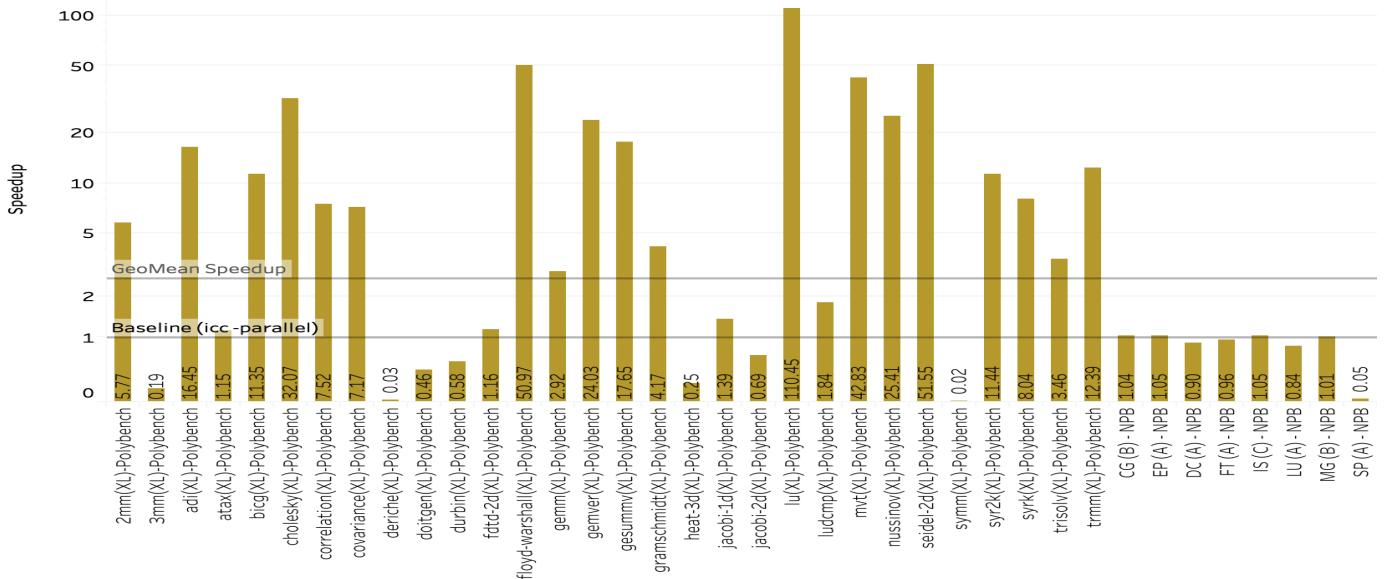


Fig. 6: *MCompiler* Speedup for Auto-Parallelized Benchmarks

auto-parallelizable loop nests in previous work. NPB benchmarks use either OpenMP or OpenACC parallel directives and therefore have potential for auto-parallelization. The directives were removed from the source code prior to processing by the *MCompiler*.

The results are shown in Figure 6. They show that the *MCompiler* improves performance over *icc* for 22 of the benchmarks. Several additional benchmarks have no change in performance. Five have a significant performance loss, which is explained in Sec. III-B4

3) *OpenMP Code*: The results are shown in Figure 7. Out of a total of 128 loop nests across all benchmarks, clang

was chosen as the most suited code optimizer for 28% loop nests, more than any other code optimizer. Loop nests that were not marked by OpenMP directives were optimized by the *MCompiler* as serial loop nests.

4) *Analysis of Results*: Analysis of the benchmarks that get slowdowns from *MCompiler*, such as 3mm, deriche (serial), symm (parallel) from Polybench and BT (serial) from NAS benchmark showed that the main reason for performance loss is the extraction of the individual loop nests that contain multiple, consecutive loop nests. Extraction of an individual loop nest inhibits code optimizers from performing loop optimizations across loop nests. Such optimizations across

loop nests, in general, improve data locality. The impact of improved data locality is even greater for multi-threaded execution.

Another reason for slowdowns can be attributed to the presence of loop nests that have a very short execution time and/or executed multiple times (in a `while` loop, for example), and performs trivial tasks such as iterating through a linked list. For such loop nests, the *MCompiler* extraction adds performance overheads.

Both of these problems can be solved in the Extractor by adding code analysis to identify consecutive loop nests and, possibly statically, identify trivial loop nests with low loop trip count. This is subject of future work.

Also, the baseline compiler, i.e. `icc`, analyzes the entire source file and can find more opportunities for optimization, including single-file interprocedural optimizations such as inlining. For OpenMP benchmarks, we did not expect much performance improvement, since code optimizers lose flexibility to optimize the OpenMP regions due to issues such as early outlining [7], [15] of code. The one exception seen in Fig. 7 is 359.botsspar, which gets a large speedup with `Polly`. The reason is the use of `Polly` optimized code for a loop nest, enclosed in a function, that is called inside a `omp task` region.

C. MCompiler with Machine Learning Prediction

This section presents experimental results for using the ML predictions for choosing the most suited code optimizer, instead of the profiling-based search in the previous sections.

1) *Machine Learning Model Training and Prediction*: Two ML models were trained, one for predicting the most suited serial, auto-vectorized code optimizer and the other one for the most suited auto-parallelizing code optimizer. Random

Forest (RF) was chosen as our classifier since the accuracy of the RF models for doing multi-class classification was better than other classification algorithms, such as Support Vector Machine (SVM). The training dataset for training the serial code classifier included loop nests from TSVC and Polybench benchmark suites and has a total of 274 instances (loop nests). The loop nests from NAS Parallel Benchmarks (NPB) were not included in the training dataset. Therefore, the experimental results for the *MCompiler* performance with ML predictions are shown for NPB benchmarks only.

The auto-parallelized code classifier was trained using the training dataset, which included loop nests from Polybench benchmark suite and has 194 instances (loop nests). Again, the experimental results for the *MCompiler* performance with ML prediction are shown for NPB benchmarks only, since these loop nests were never seen by the ML model. The reason for choosing benchmark suites such as Polybench and TSVC for creating the training dataset was to expose the ML models to a diverse set of loop nests that exhibit different characteristics. The specifics for creating the training datasets, characteristics of the training dataset and evaluating the models are similar to the work of Shivam et. al. [37].

The properties of the trained RF classifier are as follows. Maximum depth of the tree was set at 25 after analyzing that the model is neither underfitting nor overfitting on cross-validation. The maximum sub-categories were set at 15. The minimum sample count at the leaf node was set at 5. Lastly, the size of the randomly selected subset of features at each tree node that are used to find the best split is set at 20.

The serial code classifier targets (e.g. most suited code optimizers) were `clang`, `gcc`, `icc` and `Polly`. The auto-parallelized code classifier targets were `icc` and `Polly`. `pgcc` was removed as a target in order to improve the accuracy of the ML models. In the training dataset, the target for the instances with `pgcc` as the most suited code optimizer were replaced by the second best code optimizer. This decision was made after analyzing and tweaking the ML models since the accuracy of the ML models was the priority. We left out source-to-source code optimizer, such as `Pluto`, as a target code optimizer since it requires another compiler to generate code and creates noise for ML models in cases where the performance benefits are not significant from the source-level transformations.

We did not train ML models to predict the most suited code optimizer for the OpenMP loop nests for primarily one reason: the performance of the OpenMP code is largely determined by the presence of OpenMP directives and clauses rather than the properties of the loop nests.

2) *Serial Code*: The performance results for ML predictions are shown in Figure 8a relative to the profiling-based search. The most predicted code optimizer was `icc` (51%), followed by `clang` (25%). The GeoMean performance loss over the profiling-based search is 3.6%. The mis-predictions from the ML classifier was found to have a larger impact on performance when most of the execution time is dominated by one or very few kernels, such as in benchmark EP and LU.

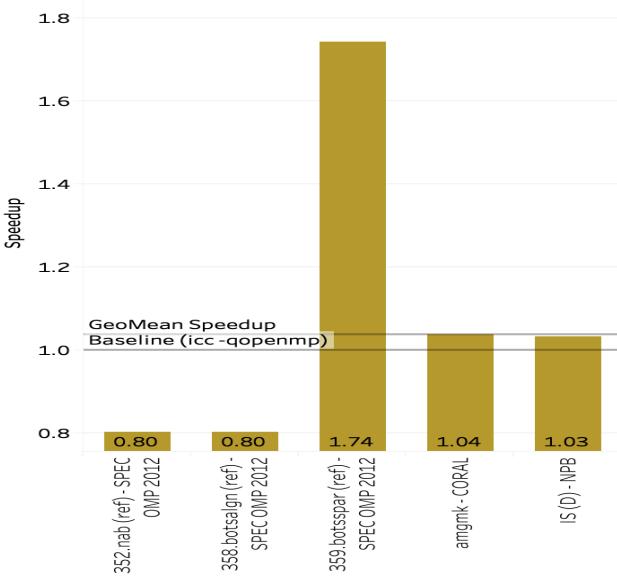


Fig. 7: *MCompiler* Speedup for OpenMP Benchmarks

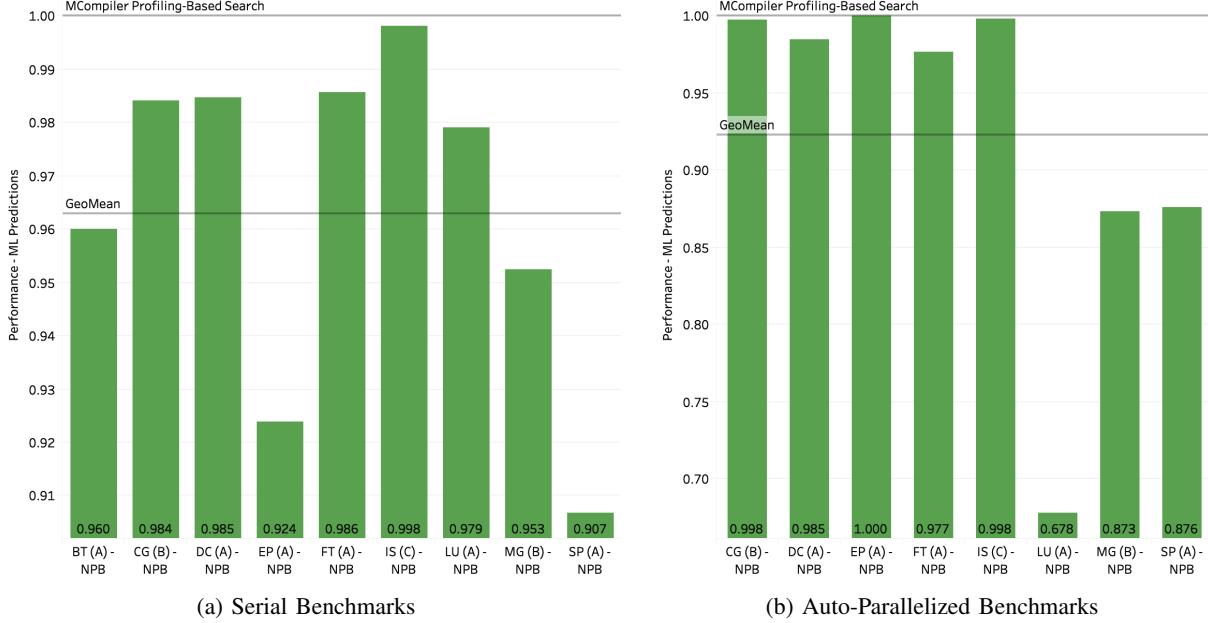


Fig. 8: *MCompiler + ML Predictions* Performance for Serial and Auto-Parallelized Benchmarks

The effect of a mis-prediction can thus be easily magnified.

3) *Auto-Parallelized Code*: The performance results for ML predictions are shown in Figure 8b relative to the profiling-based search. The most predicted code optimizer was `polly` (64%) and the rest was `icc` (36%). The impact of mispredictions is, in general, higher for auto-parallelized code as compared to serial code. Still, the GeoMean performance loss over the profiling-based search is rather small - 7.8%.

IV. RELATED WORK

Prior works such as the OptiScope infrastructure presented by Moseley et. al. [28] perform function-level and loop-level quantitative comparisons of application compiled by different compilers and/or optimization settings. Similar to our work, they look at the impact of interaction of optimization techniques for complex target architectures. But their tool performs binary analysis with the goal of assisting compiler developers in discovering new opportunities and evaluate changes. The work by Fursin et. al. [16] presents an auto-tuning framework that predicts the good combinations of optimizations to improve execution time. Their tool explores `gcc` and its optimization flags and uses ML techniques to predict good optimizations based on program features. Another work, OpenTuner framework by Ansel et. al. [2], searches for the best performing configurations for the domain-specific applications. Compared to their work, our search space is confined only to the available code optimization candidates and does not require complex heuristics or techniques to find the best performing option.

Prior works that have addressed challenges in compiler optimizations using Machine Learning have focused on auto-vectorization [39], [43] and on scalability and scheduling

configurations for the parallelism [6], [40], [42]. Stock et. al. [39] developed a ML-based performance model to guide SIMD compiler optimizations for vectorizing tensor contraction computations. However in this work, we explore kernels from a variety of computations to predict an optimizer that can generate an efficient serial code, which includes auto-vectorized code. Watkinson et. al. [43] in their work use ML models to predict opportunities for auto-vectorization and its profitability across multiple compilers and architectures. Their work also uses hardware performance counters as ML features and predict opportunities for manual vectorization in loop nests that were not auto-vectorized by the compilers. In this work, we too explore multiple code optimizers, but we are not concerned with performance gains just related to vector code generation. We apply ML on a coarser level, in order to predict the most suited code optimizer for serial as well as parallel code. Tournavitis et. al. [40] use a mix of static and dynamic features to develop a platform-agnostic, profiling-based parallelism detection method for sequential applications. Their method requires user's approval for parallelization decisions that cannot be proven conclusively. They use ML models to judge the profitability on parallelization and to select the scheduling policy. In contrast, our work uses just the dynamic features to train ML models and we let the ML models choose the most suited candidate that can generate a profitable auto-parallelized code. In future, we can incorporate the mechanism to predict number of threads and select the scheduling policy as well.

V. SUMMARY

This work presented a compilation framework, called the *MCompiler*, that optimizes application hotspots for achieving

better performance over state-of-the-art compilers. The framework incorporates optimized loop nest code - serial code, auto-parallelized code or OpenMP code - from a collection of state-of-the-art code optimizers to generate a single executable. The framework can be used with a profiling-based search to choose the most suited code optimizer for the loop nests. Experimental results showed that using the *MCompiler* with a collection of six code optimizers can significantly improve application performance.

The work also showed that one can replace the profiling-based search with an efficient Machine Learning based prediction for the most suited code optimizer for each loop nest. The results show that the Machine Learning models can predict the most suited code optimizer with a small performance loss compared to the profiling-based search. The results also show that the hardware performance counters can capture the inherent characteristics of the loop nests and the Machine Learning models based on them make good decisions.

This framework is a tool for compiler researchers to incorporate and analyze the performance of their code optimization techniques and also compare to other code optimizers.

MCompiler framework is designed to be extendable with more code optimizers, optimizer flag combinations and more features.

REFERENCES

- [1] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, Oct. 1987.
- [2] J. Ansel and et. al. Opentuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 303–315, Aug 2014.
- [3] A. H. Ashouri and et. al. MiCOMP: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):29, 2017.
- [4] D. H. Bailey and et. al. The nas parallel benchmarks summary and preliminary results. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 158–165, Nov 1991.
- [5] U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.
- [6] B. J. Barnes and et. al. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 368–377. ACM, 2008.
- [7] A. Bataev, A. Bokhanko, and J. Cownie. Towards openmp support in llvm. 2013.
- [8] U. Bondhugula and et. al. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146. Springer, 2008.
- [9] U. Bondhugula and et. al. A practical and fully automatic polyhedral program optimization system. In *ACM SIGPLAN PLDI*, 2008.
- [10] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 98–105, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
- [11] R. Cammarota and et. al. Optimizing program performance via similarity, using a feature-agnostic approach. In *Advanced Parallel Processing Technologies*, pages 199–213, Berlin, Heidelberg, 2013. Springer.
- [12] J. Cavazos and et. al. Rapidly selecting good compiler optimizations using performance counters. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 185–197. IEEE, 2007.
- [13] Z. Chen and et. al. LORE: A loop repository for the evaluation of compilers. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 219–228, Oct. 2017.
- [14] A. Darte, Y. Robert, and F. Vivien. *Scheduling and automatic Parallelization*. Springer Science & Business Media, 2012.
- [15] J. Doerfert and H. Finkel. Compiler optimizations for openmp. In *International Workshop on OpenMP*, pages 113–127. Springer, 2018.
- [16] G. Fursin and et. al. Milepost GCC: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.
- [17] Z. Gong and et. al. An empirical study of the effect of source-level loop transformations on compiler stability. *Proc. ACM Program. Lang.*, 2(OOPSLA):126:1–126:29, Oct. 2018.
- [18] T. Grosser, A. Groesslinger, and C. Lengauer. Polly - Performing Polyhedral Optimizations on a Low-level Intermediate Representation. *Parallel Processing Letters*, 22(04), 2012.
- [19] T. K. Ho. Random decision forests. In *Document analysis and recognition, 1995., proceedings of the third international conference on*, volume 1, pages 278–282. IEEE, 1995.
- [20] Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide, 2019.
- [21] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.
- [22] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 391–405. Springer, 1992.
- [23] A. W. Lim, G. I. Cheong, and M. S. Lam. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *Proceedings of the 13th International Conference on Supercomputing, ICS ’99*, pages 228–237, New York, NY, USA, 1999. ACM.
- [24] A. W. Lim and M. S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Partitions. *Parallel Comput.*, 24(3-4):445–475, May 1998.
- [25] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, PPoPP ’01*, pages 103–112, New York, NY, USA, 2001. ACM.
- [26] S. Maleki and et. al. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 372–382, Oct 2011.
- [27] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications, AIMA ’02*, pages 41–50, London, UK, UK, 2002. Springer-Verlag.
- [28] T. Moseley, D. Grunwald, and R. Peri. Optiscope: Performance accountability for optimizing compilers. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO ’09*, pages 254–264, Washington, DC, USA, 2009. IEEE Computer Society.
- [29] OpenCV (open source computer vision library), version 4.0.0. <https://opencv.org>, 2018.
- [30] The OpenMP application programming interface, version 5.0. <https://www.openmp.org/>, 2018.
- [31] Padua, Kuck, and Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, Sept 1980.
- [32] D. A. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.
- [33] PLUTO: An automatic parallelizer and locality optimizer for affine loop nests, 2015.
- [34] Polly: LLVM Framework for High-Level Loop and Data-Locality Optimizations, 2018.
- [35] PolyBench/C 4.1. <http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>, 2015.
- [36] D. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10:215–226, 2000.
- [37] A. Shivam and et. al. Towards an achievable performance for the loop nests. In *Languages and Compilers for Parallel Computing (LCPC 2018)*, 2019.
- [38] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization, CGO ’05*, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

- [39] K. Stock, L.-N. Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):50, 2012.
- [40] G. Tournavitis and et. al. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.
- [41] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216, Sep. 2010.
- [42] Z. Wang and M. F. O’Boyle. Mapping parallelism to multi-cores: A machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’09, pages 75–84, New York, NY, USA, 2009. ACM.
- [43] N. Watkinson and et. al. Using hardware counters to predict vectorization. In *Languages and Compilers for Parallel Computing (LCPC 2017)*. Springer, In Press.
- [44] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [45] R. Xu and et. al. Nas parallel benchmarks for gpgpus using a directive-based programming model. In *Languages and Compilers for Parallel Computing*, pages 67–81, Cham, 2015. Springer International Publishing.