A Multiple Compiler Framework for Improved Performance

Aniket Shivam[1], Alexandru Nicolau[2], and Alexander V. Veidenbaum[2]

[1] NVIDIA, Santa Clara, CA, USA
(Work done while at University of California Irvine, Irvine, CA, USA)
ashivam@nvidia.com
[2] University of California Irvine, Irvine, CA, USA
nicolau,alexv@ics.uci.edu

Abstract. Production compilers have achieved a high level of maturity in terms of generating efficient code. The code generated by any two production compilers can turn out to be very different based on pros and cons of their respective Intermediate Representation (IR), implemented loop transformations and their ordering, cost models used and even instruction selection (such as vector instructions) and scheduling. Hence, the performance of produced code for a program segment by a given compiler may not necessarily be matched by other compilers.

This paper proposes a meta-compilation framework, the *MCompiler*, which allows different segments of a program to be compiled with different compilers/optimizers and combined into a single executable. It turns the differences between compilation processes and performance optimizations in each compiler from a weakness to a strength. Utilizing the highest performing code for each segment can lead to a significant overall improvement in performance. A loop nest is used as a segment in this work, but other choices can be made.

The question is, though, which compiler will produce the best code for a segment. This work then presents a technique to accomplish this using Machine Learning. It learns inherent characteristics of loop nests and then predicts during compilation which code optimizer is the most suited for each loop nest in an application.

The results show that our framework improves the overall performance for applications over state-of-the-art compilers (compiled at equivalent of `-O3`) by a geometric mean of 1.97x 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 1.13x. The use of Machine Learning prediction achieves performance very close to the exploratory search for choosing the most suited code optimizer: within 4% for auto-vectorized code and within 8% for auto-parallelized code.

Keywords: Compiler Optimizations, Loop Transformations, Compilation Framework, Machine Learning

1 Introduction

An important compiler task is optimizing applications for better performance on target architectures. The means to reach the goal of producing high performance code may, and in most cases do, differ between any two production compilers. For program segments, such as loop nests, the performance of generated code from a compiler may either turn out to be better or worse compared to other compilers. They are the unavoidable result of many NP-Hard or NP-Complete problems encountered in the compilation/optimization process[23,44]. Compilers try to approximately solve NP-Hard problems efficiently and effectively by using cost models that are based on many assumptions. Compiler writers try to find the optimal solutions, based on experimentation, that work well for a large portion of target applications for their compiler, but not all. Therefore, it is quite apparent why different compilers produce different results for a given program segment. This calls for a strategy to harness the strengths of multiple compilers, while substituting the weakness of individual compilers. Hence, we are presenting a compilation framework that will provide both the users of compilers and compiler writers a means to find best possible solution for their target applications.

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, but with a caveat of additional complexity in the decision making for the compiler. The code optimization techniques to *auto-vectorize* the loop nests [35,1,52], so as to generate SIMD instructions, require careful analysis of data dependences, memory access patterns, etc. Several auto-parallelization techniques [34,26,28,27,29,25,6,12] and directive based parallel programming models, such as OpenMP [33], 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 [21,37,38].

Code optimizers apply a 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 NP-Hard 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. A major challenge in developing the heuristics and cost 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 [32,47,30,16] 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. From a given code optimizer's point of view, the sequence it used is the best it could do 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 for it. The *MCompiler* identifies loop nests in `C` applications, optimizes the loop nests using different code optimizers, times each optimized code version in execution of its complete application, and links the best performing code to generate the complete application binary. This is referred to as the *exploratory search* method of the *MCompiler*. The *MCompiler* currently incorporates code optimizers from Intel's C compiler, GNU GCC, LLVM Clang and PGI's C compiler. In addition to these, two Polyhedral Model based loop optimizers, Polly [17,38] and Pluto [7,37] are used, if applicable. The best loop nest code selection allows the *MCompiler* to produce higher-performing code than the best of the code optimizers in the framework. The *MCompiler* benefits from the entire compilation process (loop transformations and optimizations, and code generation) implemented in each of the code optimizers. The paper presents a study of the proposed approach's potential. It optimizes each extracted loop nest separately with all available code optimization *candidates*. The performance of each optimized loop nest is measured as part of the complete application execution. The best performing code for a loop nest is selected for linking into the final executable. This step, referred to as exploratory search, shows that the framework can indeed improve the resulting code's performance. We show that our framework achieves an overall geometric mean speedup of 1.97x for serial code, 2.62x for auto-parallelized code and 1.13x for OpenMP code over Intel C Compiler (compiled with `-Ofast`) across benchmarks from multiple benchmark suites.

However, this prompts the question "Can we learn and predict which compiler will produce the best code for a loop nest without an expensive search?". The paper proposes a Machine Learning (ML) based technique to predict the most suited code optimizer for a given loop nest. This makes the use of the exploratory search step in the framework unnecessary for selecting the best compiler for a loop nest. However, as with any prediction, it can lead to a potential performance loss compared to search-based selection due prediction errors, e.g., when the ML model or classifier does not choose the best code optimizer. Our results show that by using well-trained ML models this potential loss in performance can be quite small. This section of our work is an extension to the work by Shivam et. al. [42].

Embedding Machine Learning models in compilers is continuously being explored by the research community [31,44,43,10,47,48,15,36,45,9,24,49,3,42,18]. Previous studies have shown that hardware performance counters can successfully capture the characteristic behavior of a loop nest. Hardware performance counters can capture intricate details about data movement across levels of caches, memory footprint, and count and types of instructions retired that determine the performance of loop nests on an architecture. The approach used in this paper relies on hardware performance counters collected for a loop nest.

In our approach the hardware performance counters are collected from a single profile of the applications, i.e., the applications are compiled with just one code optimizer and then executed once. A number of possible Machine Learning features were investigated. The best results were achieved using the hardware performance counter data collected from profiling a serial (`-O1`) version of a loop nest. This is what is used in this paper. The reason for using `-O1` version of the loop nest is that this version shows inherent code characteristics, i.e., generated code is an unoptimized version without any complex code transformations.

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 exploratory search for the most suited code optimizer. We skip ML predictions for loop nests parallelized using OpenMP directives because in this case code optimizers lose flexibility to optimize the OpenMP regions and the performance is no longer just dependent on the inherent characteristics of loop nests. Hence, this problem is not suitable for such predictions.

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 shows that using the framework can achieve better performance over state-of-the-art compilers (compiled at equivalent of -Ofast) by a geometric mean of 1.97x for auto-vectorized code, 2.62x for auto-parallelized code and 1.13x for OpenMP code.
– 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 (with under 4% performance loss).
– The framework will be open sourced for researchers and compiler developers to analyze and compare their code optimization techniques.

The rest of the paper is organized as follows. Section 2 describes the *MCompiler* framework and the methodology for choosing the most suited code optimizer for a loop nest using exploratory search as well as using ML-based prediction. Section 3 describes the evaluation methodology and analyzes the experimental results. Section 4 discusses related work. We conclude the paper with Section 5.

2 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. This framework achieves significant performance improvements as will be seen in the Experimental Analysis section. The changes for incorporating Machine Learning predictions are discussed later in the next section.

2.1 Overall Framework Architecture

Figure 1 shows the structure of the *MCompiler* framework. The first phase is the *Loop Extractor* from `C` applications. The *Extractor* parses the source files

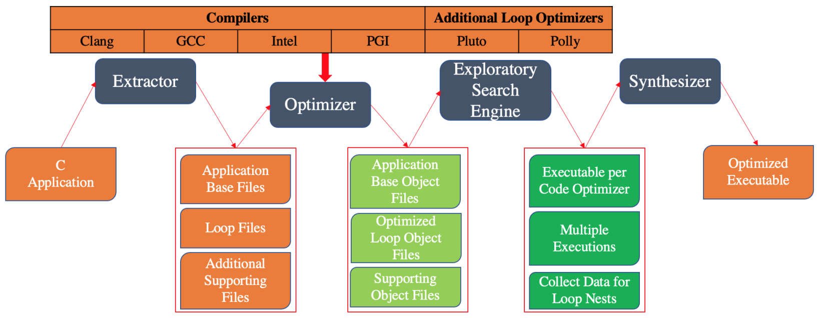

Compilers
Additional Loop Optimizers
Clang
GCC
Intel
PGI
Pluto
Polly
Extractor
Optimizer
Exploratory Search Engine
Synthesizer
C Application
Application Base Files
Loop Files
Additional Supporting Files
Application Base Object Files
Optimized Loop Object Files
Supporting Object Files
Executable per Code Optimizer
Multiple Executions
Collect Data for Loop Nests
Optimized Executable

Fig. 1: *MCompiler* Framework

to find loop nests, extract those loop nests as `functions` into separate, independently compilable files and replaces 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. [11] 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 *Exploratory Search* phase, where an application is executed to record the execution times of the extracted loop nests. Executables generated for each code optimizer are 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 from every code optimizer are compared and the best performing code/optimizer is selected, i.e., the optimized code that executes the loop body in the shortest time. Finally, the default compiler links the selected object files for every loop nest 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.

2.2 Loop Extraction Phase

The loop extractor works in three phases and is implemented using `ROSE`, a source-to-source compiler infrastructure [40]. 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 adds `extern` declarations for global variables and global functions, as well as for functions called in the scope of the loop body. It 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 it replaces the loop body with a function call (with required arguments) and adds an `extern` declaration to this function. Finally, it generates 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 versions for each loop file, where one version is instrumented to collect the execution time for the loop nest. This version is used during the Exploratory phase. The other version does not contain any instrumentation code and is used to generate the final executable for the applications.

Function Definition enclosing the Loop Nests The extractor generates the list 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 copies the function parameters of primitive types (passed by reference) into local variables (with same names as original variables) before the loop body and correspondingly copies 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.

The extractor also annotates loop nests with `pragma scop/endscop` so as to aid source-to-source Polyhedral optimizers, such as Pluto, in locating Static Control Parts (SCoP). If the loop nest was indeed not a SCoP, then Polyhedral optimizers can't optimize them. The framework will recognize that in the Optimization Phase and discard Polyhedral optimizers as a candidate for those loop nests. 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 issue with 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.

2.3 Optimization Phase

The framework currently 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 a source-to-source loop optimizer, i.e., Pluto. The flags used for optimizing loop nests are equivalent of `-Ofast` for serial execution, and `-parallel` for parallel execution. For polyhedral loop optimizers, flags to enable tiling, vectorization and auto-parallelization are selected. 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.

2.4 Exploratory Search Phase

The Exploratory Search Engine invokes executables generated by the code optimizers one-by-one and performs multiple runs for stable data, if requested. Exploratory Search Engine at the end of each execution collects the information for each of the loop nests and forwards it to the Synthesizer. For applications that need input through command line, the Exploratory Search Engine runs the application with the input given to the *MCompiler* framework.

2.5 Synthesis Phase

The synthesizer compares the collected execution times for each loop nest from different code optimizers and chooses the code optimizer that performed the best as the most suited code optimizer. For loop nests with no information, i.e., the code that was not executed during Exploratory Search phase, the default compiler is used. The synthesizer then generates the final executable that contains no instrumentation code. For an OpenMP application, the synthesizer links OpenMP runtime libraries that are used by different compilers, e.g., `icc`, `pgcc` and `clang` use compatible OpenMP runtime libraries whereas `gcc` doesn't. Therefore, say, if for an application *MCompiler* chooses a `omp parallel for` region from `icc` and another from `gcc`, then the parallel regions will be executed by different OpenMP runtime libraries. Static libraries specific to compilers are also linked to successfully generate the final executable.

2.6 Framework Architecture for Machine Learning Predictions

The framework for choosing the most suited code optimizer for loop nest using ML prediction is shown in figure 2. The goal is to eliminate the time-consuming Exploratory Search step of the framework and use the ML prediction to select the best code optimizers during compilation. 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 a prediction are the hardware performance counter values collected during execution of a loop nest. The use of hardware performance counter-based measurements to predict the most suited code optimizer follows the work of Shivam et. al. [42]. Prior work [31,43,10,47,48,15,36,45,9,24,49,3] have used Machine Learning in compilers for various tasks, such as, selecting the best performing flag combinations, predicting the best loop unrolling factor, selecting the best polyhedral optimizations, predicting the likelihood of vectorization for loop nests, predicting about parallelism in loop nests, etc. The work of Shivam et. al. [42] was the first to show the possibility of predicting the best code optimizer for loop nests.

We empirically chose Random Decision Forest [20] as our classification algorithm to predict the most suited code optimizer for the loop nests. Prior work that automatically generated features from the compiler's intermediate representation (IR) [24] has shown that Machine Learning algorithms do learn from features that may not be intuitive even to an expert compiler writer. Most helpful features for a Machine Learning algorithms are difficult to explain precisely, but understanding them can help infer their importance. We looked at the most important features for our Machine Learning classifier and made an attempt at inferring their importance for the classifier.

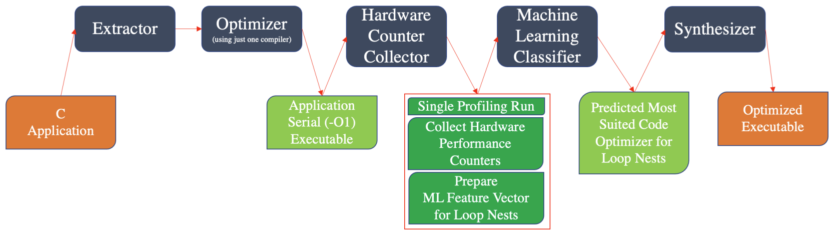

Extractor
Optimizer
(using just one compiler)
Hardware Counter Collector
Machine Learning Classifier
Synthesizer
C Application
Application Serial (-O1) Executable
Single Profiling Run
Collect Hardware Performance Counters
Prepare ML Feature Vector for Loop Nests
Predicted Most Suited Code Optimizer for Loop Nests
Optimized Executable

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

Some of the key reasons for why hardware performance counters as features are able to capture the characteristic behavior of loop nests are as follows. First, instruction count and stalls related to data movement such as load and store instructions retired, either scalar or vector, characterize traffic effect from the TLBs, caches and RAM. Second, stall cycle counters (for hardware resources) determine if the loop nest performance is limited by a particular resource. Third, L1, L2 and L3 hit/miss counters determine the memory footprint and provide information about the data access pattern. For example, the counters for a loop nest with a stride-1 access pattern have lower L1 and/or L2 cache misses than a loop nest with larger or non-linear strides. Stride-1 access also correlates with vectorizability. Also, the hardware prefetchers are stride 1 or next line, resulting in further latency reduction. Lastly, instructions per cycle retired for arithmetic operations on different data types such as Floating Point (both single and double precision) or Integer provide information about the throughput/latency of computations.

The code optimizers perform a set of optimizations/loop transformations that are based on properties of the loop nest. For example, different code optimizers may choose to unroll the innermost loop by different factor or just not unroll at all based its loop trip count and memory access pattern. Such properties of the loop nests are captured by the hardware counters. Similarly, a lot of cache misses at L1 and L2 level, may suggest transformations like loop interchange or loop tiling can be beneficial. Another example would be if L1 and L2 have higher cache misses than L3, then loop interchange, if possible, could benefit towards getting better performance by allowing much more efficient access to data. Transformations, such as unrolling, tiling and interchange, may also lead to vector code generation that often has high impact on performance. So counters do correlate with potentially beneficial transformations and one compiler may perform such transformations or combinations thereof better than another.

The architecture of the *MCompiler* framework is modified in the following ways for ML prediction of the most suited code optimizer for a loop nest. First, the Optimizer now generates an executable that is compiled by the default compiler for serial execution with `-O1` optimization level. Second, the Exploratory Search Engine is replaced by the Hardware Counter Collector for making ML predictions. The hardware Counter Collector executes the serial (`-O1`) code and collects hardware performance counters for each loop nest. This is done only once using the default compiler, in contrast to the exploratory search that has to run every compiler. As mentioned earlier, using the `-O1` version of the loop nest helps in preserving the inherent code characteristics, since the generated code is not optimized using complex code transformations, and is similar across compilers, therefore providing a good common baseline. If a loop nest is not executed or the hardware performance counters are not present (e.g. 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 from the predicted optimizer and links the selected optimized loop object files and generates the final executable for the application.

Collecting Hardware Performance Counters for the Loop Nests We use the Intel compiler to generate the executable that is then used for profiling. All loop optimizations are disabled during this compilation by using the `-O1` flag. In addition to that, the optimizations 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 and memory-based (D-TLB, L1 cache, L2 cache, L3 cache) counters. Once the hardware performance counters are collected for the loop nests, the dynamic instruction count is not used as a feature. The other hardware performance counters are normalized 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.

3 Experimental Analysis

This section describes the experimental methodology and present the results and their analysis demonstrating the effectiveness of the *MCompiler* framework.

3.1 Benchmarks, Code Optimizers and Target Architecture

Several different benchmark suites are used to evaluate the effectiveness of the *MCompiler* framework. One is Test Suite for Vectorizing Compilers (TSVC) by Callahan et al. [8] and Maleki et al. [30]. 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 [39]. 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 [53]. These benchmarks are used in serial code, auto-parallelized code and OpenMP parallel code experiments. Lastly, a set of `C` benchmarks from Parboil [46] and SPEC OMP 2012 were used for OpenMP experiments. The `train` dataset was used for SPEC benchmarks during the exploratory search phase, whereas the results are shown for `ref` dataset. Six code optimizers are incorporated in the *MCompiler* framework, as mentioned in section 2.3. 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 as shown in Fig. 5. 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 [14].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.

3.2 *MCompiler* with Exploratory Search

This section presents results of the exploratory search by the *MCompiler* for choosing the most suited code optimizer for three benchmark suites: complete TSVC, Polybench, and NAS Benchmark Suite (NPB). Each application was executed 3 times for each of the code optimizers and the median execution time was chosen for deciding the most suited code optimizer.

Serial Code The results are shown in Fig. 3. The benchmark labels show 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`. As shown in Fig. 5, `icc` was chosen as the most suited code optimizer for 49% of the loop nests, followed by `Pluto` (source-to-source optimizer, compiled with `icc`) at 22.5%. In many of those 22.5% cases, loop tiling from `Pluto` followed by vector code generation from `icc` provided better performance than just `icc` itself.

The performance of the *MCompiler* for Polybench benchmarks is 2.44x (GeoMean) better than `icc`. As expected, the two polyhedral model based optimizers

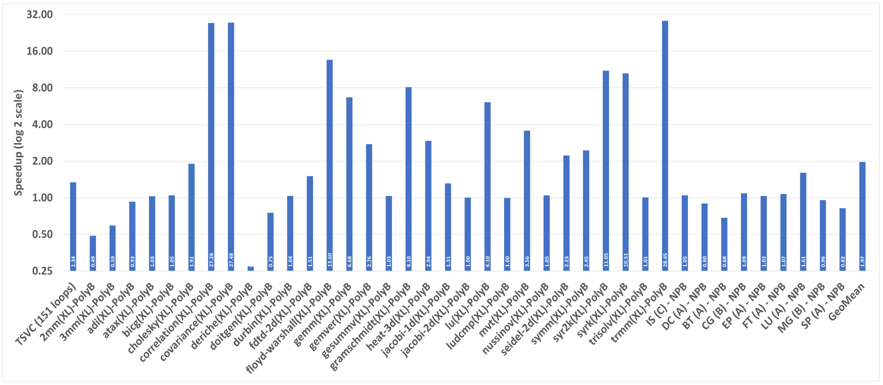


Fig. 3: *MCompiler* Speedup for Serial Benchmarks

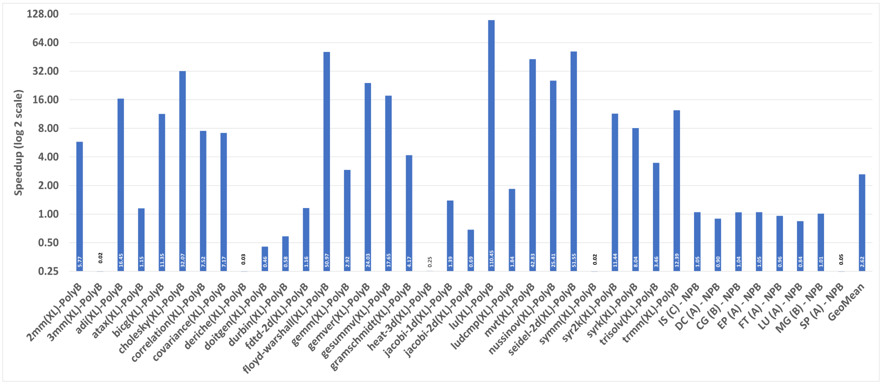


Fig. 4: *MCompiler* Speedup for Auto-Parallelized Benchmarks

were chosen as the most suited code optimizer for 60% of the loop nests that dominate execution time of the main kernels for Polybench benchmarks. `icc` and `clang` were each chosen as the most suited code optimizer for 14.6% of the kernel loop nests, with the remaining loop nests split equally among the remaining code optimizers. `icc` was chosen as the most suited code optimizer for 153 out of 288 (53%) loop nests from NPB benchmarks (not counting loop nests such as array initialization loops).

Overall the percentage of loop nests chosen from each code optimizer can be seen in Fig. 5. For analysis shown in Fig. 5, we removed trivial loop nests that perform tasks that do not test the optimization capabilities of the code optimizers. For example, loop nests that are used to allocate dynamic memory, to perform array initialization, etc. It shows that across all benchmarks, while `icc` dominates overall, 52% of loop nests are best optimized by other code optimizers (with approximately equal distribution among them, except for `gcc`). More details for specific cases are explained in section 3.2.

Auto-Parallelized Code These experiments were performed with 32 threads for both the exploratory search phase 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.

The results in Fig. 4 show that the *MCompiler* improves performance over `icc`, by at least 5%, for 23 out of 38 benchmarks. Several additional benchmarks have no change in performance. Five have a significant performance loss, which is explained in section 3.2. Overall the percentage of loop nests chosen from each code optimizer can be seen in Fig. 5. Similar to the trend seen for serial code benchmarks, `icc` dominates for NPB benchamrks, whereas polyhedral model based optimizers perform better for Polybench benchmarks.

	TSVC		Polybench		NPB-SER		Total	
	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage
Clang	21	13.9%	7	14.6%	27	9.4%	55	11.3%
GCC	3	2.0%	2	4.2%	26	9.0%	31	6.4%
ICC	74	49.0%	7	14.6%	153	53.1%	234	48.0%
PGCC	10	6.6%	3	6.3%	39	13.5%	52	10.7%
Pluto*	34	22.5%	17	35.4%	8	2.8%	59	12.1%
Polly	9	6.0%	12	25.0%	35	12.2%	56	11.5%

(a) Serial Code

	Polybench		NPB-OMP/ACC		Total	
	Loop Nests	Percentage	Loop Nests	Percentage	Loop Nests	Percentage
ICC	2	4.2%	229	72.5%	231	63.5%
PGCC	5	10.4%	56	17.7%	61	16.8%
Pluto*	35	72.9%	8	2.5%	43	11.8%
Polly	6	12.5%	23	7.3%	29	8.0%

(b) Auto-Parallelized Code

* Pluto optimized code was compiled with ICC

Fig. 5: Distribution of best performing code per Code Optimizer. Breakdowns per benchmarks suite showcase benefits of specialized code optimizers.

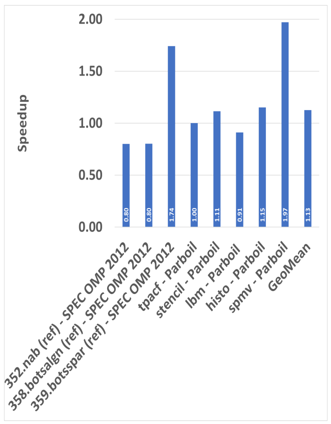


Fig. 6: *MCompiler* Speedup for OpenMP Benchmarks

OpenMP Code The results are shown in Fig. 6. Loop nests that were not marked by OpenMP directives were optimized by the *MCompiler* as serial loop nests. We did not expect much performance improvement from OpenMP regions, since code optimizers lose flexibility to optimize the OpenMP regions due to issues such as early outlining [5,13] of code. The results show that in a few cases high speedups can indeed be achieved using the *MCompiler*. The reason for such performance gains is explained in section 3.2.

Analysis of Results Analysis of the benchmarks that get slowdowns from the *MCompiler*, such as 2mm (serial), 3mm (serial), deriche (serial), symm (parallel) from Polybench, and BT (serial) and SP (parallel) from NAS benchmark showed that the main reason for performance loss, based on compiler generated reports, is the early outlining of loop nests into individual functions. This may hinder the alias analysis and therefore compilers may generate sub-optimal code for such loop nests.

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 perform trivial tasks such as iterating through a linked list. For such loop nests, the *MCompiler* extraction adds performance overheads.

This problem can be solved in the Extractor by automatically identifying trivial loop nests with low loop trip count. This is subject of future work. For now, if users want to manually mark such trivial loop nests with low loop trip count, they can add `pragma MC skiploop` directive. Although, we didn't add such directives or do any modificiation to the original source codes for the experiments presented in this 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.

Fig. 6 shows speedups for OpenMP benchmarks. It shows significant speedups for *MCompiler* on `359.botsspar` from `SPEC OMP 2012` and `spmv` from `Parboil`. The reason for speedup in `359.botsspar` is a loop nest with a computation similar to matrix multiplication that is enclosed in a function, that is called inside a `omp task` region. The *MCompiler* optimized this particular loop nest as it do would for a serial loop nest and chose `Polly` generated code for this loop nest. The reason for speedup in `spmv` is that, inside the `OMP parallel for` region, the inner most loop was vectorized and unrolled by `gcc`. Whereas, `icc` chose to not generated vector code for the same loop nest because, as per the compiler generated report, the cost models suggested slowdowns because of vectorization. The inner most loop had irregularly indexed loads which seemed to have impacted the decision of `icc`'s cost models.

Key factors contributing to performance difference between code optimizers, other than their loop transformations, are as follows. First, a difference in unroll factor, which leads to the difference in type of vector instructions selected and also leads to more consecutive load/store of data. Second, generation of multi-variant code, which chooses the best code during execution based on runtime analysis of dependences. Third, use of specialized libraries, such as the vectorized math library.

3.3 *MCompiler* with Machine Learning Prediction

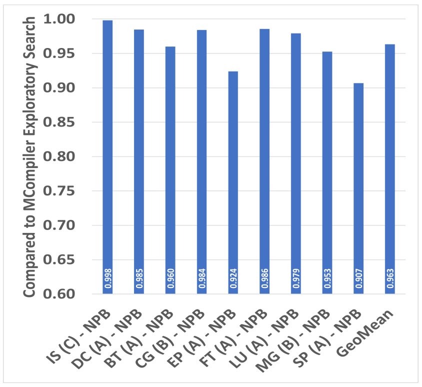
The training dataset for training the serial code classifier included loop nests from TSVC and Polybench benchmark suites and had a total of 274 instances (loop nests). The loop nests from NAS Parallel Benchmarks (NPB) were not included in the training dataset. Therefore, the experimental results for 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 not used in training 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. [42].

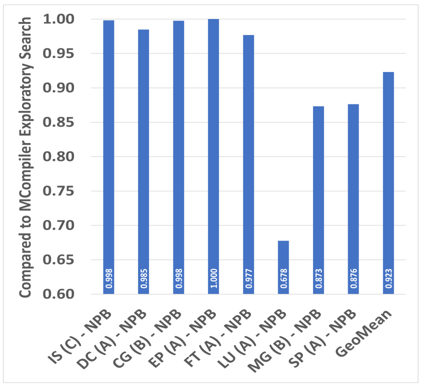
The properties of the trained RF classifier are as follows. Maximum depth of the tree was set at 25 after analyzing that the model was neither underfitting or overfitting on cross-validation. The maximum sub-categories were set at 15. The minimum sample count at the leaf node was set at 5. Lastly, the size of the randomly selected subset of features at each tree node that are used to find the best split was 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 tuning and analyzing the ML models on two performance measures: the area under the Receiver Operating Characteristic (ROC) curve, also known as AUC, and classification accuracy. The evaluation of ML models showed that replacing `pgcc` as a target class improved both AUC and classification accuracy. Two reasons why this strategy worked in improving the ML models are as follows. First, the lack of enough instances in the training dataset with target class being `pgcc`, i.e., not enough loop nests with best code optimizer being `pgcc`. The training dataset is skewed towards `icc` as the target class due to `icc` being chosen as the most suited code optimizer, similarly shown in Shivam et. al. [42]. This leads to an unbalanced dataset where minority classes have a very small share of the training instances and therefore are a major challenge for ML classifiers [50,22,51,19]. Removing `pgcc` improved the share of other minority classes such as `clang`, `gcc` and `Polly`. Hence, improving both performance measures, AUC and classification accuracy, significantly. Second, on analyzing the dataset we found that for a lot of instances where `pgcc` was chosen as the best code optimizer, it had a small margin over the second best code optimizer. This can potentially lead to another challenge for ML classifiers, that is high overlapping in the feature space for multi-class classification.

We left out source-to-source code optimizer (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. Since `Polly` optimizations/passes run along with the standard `LLVM` pass pipeline, `Polly` is considered as the most suited code optimizer only when it shows at least 5% performance improvement over `clang`. There are two reasons for setting a 5% performance improvement threshold before attributing a loop nest to `Polly` over `clang`. First, from an ML point-of-view, if there are two very similar loop nests (that will lie very close in a multi-dimensional feature space), one has `Polly` as the target class whereas the other has `clang`, then a ML algorithm will try to overfit in order to reduce the classification error, while the actual impact on the performance will be quite minimal. Second, a 5% execution time difference could just be a time variance between multiple runs, i.e., experimental error. We did not train ML models to predict the most suited code optimizer for the OpenMP regions for primarily one reason: the code optimizers lose flexibility to optimize the OpenMP regions as mentioned before.



(a) Serial Benchmarks



(b) Auto-Parallelized Benchmarks

Fig. 7: *MCompiler* + ML Predictions Performance for Serial and Auto-Parallelized Benchmarks

Serial Code The performance results for ML predictions are shown in Fig. 7a relative to the exploratory search. The most predicted code optimizer was `icc` (51%), followed by `clang` (25%). The GeoMean performance loss over the exploratory search is 3.6%. The mis-predictions from the ML classifier were found to have a larger impact on performance when most of the execution time is dominated by one or very few kernels. The effect of a mis-prediction can thus be significantly magnified. One such case is the EP benchmark where 88% of the execution time is spent in one loop nest and `clang` was mis-predicted as the most suited code optimizer for that loop nest instead of `icc`. For SP benchmark, the exclusion of `pgcc` from ML predictions is responsible for the 9% performance loss compared to the exploratory search which included all available code optimizers.

Auto-Parallelized Code The performance results for ML predictions are shown in Fig. 7b relative to the exploratory search. The most predicted code optimizer was `polly` (64%) and the rest was `icc` (36%). For LU benchmark, `Polly` was predicted as the most suited code optimizer for multiple loop nests used in computing the right hand side (rhs), whereas `icc` was chosen as the most suited code optimizer by the exploratory search for those loop nests. For MG and SP benchmarks, the exclusion of `pgcc` from ML predictions is responsible for the 12% performance loss compared to the exploratory search. The impact of mis-predictions is, in general, higher for auto-parallelized code as compared to serial code. Still, the Geometric Mean of performance loss over the exploratory search is rather small - 7.8%.

4 Related Work

Prior work such as the work by Fursin et. al. [15] presents an auto-tuning framework that predicts good combinations of compiler flags to improve execution time. Their tool explores `gcc` and its flags and uses ML techniques to predict good optimizations based on program features. Another similar work, the `OpenTuner` framework by Ansel et. al. [2], searches for the best performing GCC/G++ flag combinations for C/C++ applications, in addition to searching configurations for Halide and other domain specific applications. The above mentioned work by Fursin et. al. and Ansel et. al. explore different combinations of code optimizer flags that has been extensively studied is orthogonal to our work. Our work used the most recommended flag combinations for each code optimizer. `NeuroVectorizer` [18] proposed an approach for handling loop vectorization and an end-to-end solution using deep reinforcement learning (RL). It finds two vectorization parameters via RL, the loop unrolling factor and the interleaving factor. The results in [18] show a 1.33x average speedup over Polly on six of the Polybench benchmarks. Interestingly, the MCompiler also had a speedup of 1.33x over Polly for these six benchmarks (even though the *MCompiler* had a 2.5x slowdown on one of the benchmarks due to aliasing issues). One can conjecture from this that additional factors, such as better loop transformations, are the reasons for *MCompiler* speedups. Alternatively, if the two vectorization parameters targeted by the `NeuroVectorizer` are indeed responsible for the performance gains, then at least one of the optimizers in the *MCompiler* framework already does it equally well. But not the same optimizer all the time. One can also incorporate tools like the `OpenTuner` and `NeuroVectorizer` into our framework and potentially obtain additional speedups.

5 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 exploratory search to choose the most suited code optimizer for the loop nests. Exploratory search results showed that the *MCompiler* with six code optimizers can significantly improve application performance.

The work also showed that one can replace the exploratory search with an efficient Machine Learning based prediction for the most suited code optimizer for a 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 exploratory search. The results also demonstrate 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 can also be used as a tool for compiler researchers to incorporate and analyze the performance of their code optimization techniques and compare to other code optimizers. *MCompiler* framework is open source[3] and is designed to be extendable with more code optimizers, optimizer flag combinations and more features. More details about this work can be found in the thesis [41].

[3]Available at https://github.com/ANIKET-SHIVAM/MCompiler

References

1. Allen, R., Kennedy, K.: Automatic translation of fortran programs to vector form. ACM Trans. Program. Lang. Syst. **9**(4), 491–542 (Oct 1987). https://doi.org/10.1145/29873.29875, http://doi.acm.org/10.1145/29873.29875
2. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U., Amarasinghe, S.: Opentuner: An extensible framework for program autotuning. In: 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT). pp. 303–315 (2014)
3. Ashouri, A.H., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., Cavazos, J.: MiCOMP: Mitigating the compiler phase-ordering problem using optimization subsequences and machine learning. ACM Transactions on Architecture and Code Optimization (TACO) **14**(3), 29 (2017)
4. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The nas parallel benchmarks summary and preliminary results. In: Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. pp. 158–165 (Nov 1991). https://doi.org/10.1145/125826.125925
5. Bataev, A., Bokhanko, A., Cownie, J.: Towards openmp support in llvm (2013)
6. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: International Conference on Compiler Construction. pp. 132–146. Springer (2008)
7. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 101–113 (2008)
8. Callahan, D., Dongarra, J., Levine, D.: Vectorizing compilers: A test suite and results. In: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing. pp. 98–105. Supercomputing '88, IEEE Computer Society Press, Los Alamitos, CA, USA (1988), http://dl.acm.org/citation.cfm?id=62972.62987
9. Cammarota, R., Beni, L.A., Nicolau, A., Veidenbaum, A.V.: Optimizing program performance via similarity, using a feature-agnostic approach. In: Advanced Parallel Processing Technologies. pp. 199–213. Springer, Berlin, Heidelberg (2013)
10. Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M.F., Temam, O.: Rapidly selecting good compiler optimizations using performance counters. In: International Symposium on Code Generation and Optimization (CGO'07). pp. 185–197. IEEE (2007)
11. Chen, Z., Gong, Z., Szaday, J.J., Wong, D.C., Padua, D., Nicolau, A., Veidenbaum, A.V., Watkinson, N., Sura, Z., Maleki, S., Torrellas, J., DeJong, G.: Lore: A loop repository for the evaluation of compilers. In: 2017 IEEE International Symposium on Workload Characterization (IISWC). pp. 219–228 (2017)
12. Darte, A., Robert, Y., Vivien, F.: Scheduling and automatic Parallelization. Springer Science & Business Media (2012)
13. Doerfert, J., Finkel, H.: Compiler optimizations for openmp. In: International Workshop on OpenMP. pp. 113–127. Springer (2018)
14. Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., Mishra, P.: The design and operation of CloudLab. In: Proceedings of the USENIX Annual Technical Conference (ATC). pp. 1–14 (Jul 2019), https://www.flux.utah.edu/paper/duplyakin-atc19

15. Fursin, G., Kashnikov, Y., Memon, A.W., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., et al.: Milepost GCC: Machine learning enabled self-tuning compiler. International journal of parallel programming **39**(3), 296–327 (2011)
16. Gong, Z., Chen, Z., Szaday, J., Wong, D., Sura, Z., Watkinson, N., Maleki, S., Padua, D., Veidenbaum, A., Nicolau, A., Torrellas, J.: 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). `https://doi.org/10.1145/3276496`, `http://doi.acm.org/10.1145/3276496`
17. Grosser, T., Groesslinger, A., Lengauer, C.: Polly - Performing Polyhedral Optimizations on a Low-level Intermediate Representation. Parallel Processing Letters **22**(04) (2012)
18. Haj-Ali, A., Ahmed, N.K., Willke, T., Shao, Y.S., Asanovic, K., Stoica, I.: NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning. In: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization. pp. 242–255. CGO 2020, Association for Computing Machinery, New York, NY, USA (2020). `https://doi.org/10.1145/3368826.3377928`, `https://doi.org/10.1145/3368826.3377928`
19. He, H., Garcia, E.A.: Learning from imbalanced data. IEEE Transactions on Knowledge and Data Engineering **21**(9), 1263–1284 (2009)
20. Ho, T.K.: Random decision forests. In: Document analysis and recognition, 1995., proceedings of the third international conference on. vol. 1, pp. 278–282. IEEE (1995)
21. Automatic parallelization with intel®compilers (August 2018), `https://software.intel.com/content/www/us/en/develop/articles/automatic-parallelization-with-intel-compilers.html`
22. Japkowicz, N., Stephen, S.: The class imbalance problem: A systematic study. Intelligent data analysis **6**(5), 429–449 (2002)
23. Kremer, U.: Optimal and near – optimal solutions for hard compilation problems. Parallel Processing Letters **7**(04), 371–378 (1997)
24. Leather, H., Bonilla, E., O'Boyle, M.: Automatic feature generation for machine learning–based optimising compilation. ACM Trans. Archit. Code Optim. **11**(1) (Feb 2014). `https://doi.org/10.1145/2536688`, `https://doi.org/10.1145/2536688`
25. Lee, S.I., Johnson, T.A., Eigenmann, R.: Cetus – an extensible compiler infrastructure for source-to-source transformation. In: Rauchwerger, L. (ed.) Languages and Compilers for Parallel Computing. pp. 539–553. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
26. Li, W., Pingali, K.: A singular loop transformation framework based on non-singular matrices. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 391–405. Springer (1992)
27. Lim, A.W., Cheong, G.I., Lam, M.S.: An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In: Proceedings of the 13th International Conference on Supercomputing. pp. 228–237. ICS '99, ACM, New York, NY, USA (1999). `https://doi.org/10.1145/305138.305197`, `http://doi.acm.org/10.1145/305138.305197`
28. Lim, A.W., Lam, M.S.: Maximizing Parallelism and Minimizing Synchronization with Affine Partitions. Parallel Comput. **24**(3-4), 445–475 (May 1998). `https://doi.org/10.1016/S0167-8191(98)00021-0`, `http://dx.doi.org/10.1016/S0167-8191(98)00021-0`

29. Lim, A.W., Liao, S.W., Lam, M.S.: 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. pp. 103–112. PPoPP '01, ACM, New York, NY, USA (2001). `https://doi.org/10.1145/379539.379586`, `http://doi.acm.org/10.1145/379539.379586`
30. Maleki, S., Gao, Y., GarzarÂ´n, M.J., Wong, T., Padua, D.A.: An evaluation of vectorizing compilers. In: 2011 International Conference on Parallel Architectures and Compilation Techniques. pp. 372–382 (Oct 2011). `https://doi.org/10.1109/PACT.2011.68`
31. Monsifrot, A., Bodin, F., Quiniou, R.: A machine learning approach to automatic production of compiler heuristics. In: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications. pp. 41–50. AIMSA '02, Springer-Verlag, London, UK, UK (2002), `http://dl.acm.org/citation.cfm?id=646053.677574`
32. Moseley, T., Grunwald, D., Peri, R.: Optiscope: Performance accountability for optimizing compilers. In: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 254–264. CGO '09, IEEE Computer Society, Washington, DC, USA (2009). `https://doi.org/10.1109/CGO.2009.26`, `http://dx.doi.org/10.1109/CGO.2009.26`
33. The OpenMP application programming interface, version 5.0. `https://www.openmp.org/` (November 2018)
34. Padua, Kuck, Lawrie: High-speed multiprocessors and compilation techniques. IEEE Transactions on Computers **C-29**(9), 763–776 (Sept 1980). `https://doi.org/10.1109/TC.1980.1675676`
35. Padua, D.A., Wolfe, M.: Advanced Compiler Optimizations for Supercomputers. Commun. ACM **29**(12), 1184–1201 (1986)
36. Park, E., Pouche, L., Cavazos, J., Cohen, A., Sadayappan, P.: Predictive modeling in a polyhedral optimization space. In: International Symposium on Code Generation and Optimization (CGO 2011). pp. 119–129 (2011)
37. PLUTO: An automatic parallelizer and locality optimizer for affine loop nests (2015), `http://pluto-compiler.sourceforge.net`
38. Polly: LLVM Framework for High-Level Loop and Data-Locality Optimizations (2018), `http://polly.llvm.org`
39. PolyBench/C 4.1. `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/` (2015)
40. Quinlan, D.: Rose: Compiler support for object-oriented frameworks. Parallel Processing Letters **10**, 215–226 (2000)
41. Shivam, A.: A Multiple Compiler Approach for Improved Performance and Efficiency. Ph.D. thesis, UC Irvine (2021)
42. Shivam, A., Watkinson, N., Nicolau, A., Padua, D., Veidenbaum, A.V.: Towards an achievable performance for the loop nests. In: Hall, M., Sundar, H. (eds.) Languages and Compilers for Parallel Computing. pp. 70–77. Springer International Publishing, Cham (2019)
43. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: Proceedings of the International Symposium on Code Generation and Optimization. pp. 123–134. CGO '05, IEEE Computer Society, Washington, DC, USA (2005). `https://doi.org/10.1109/CGO.2005.29`, `http://dx.doi.org/10.1109/CGO.2005.29`
44. Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.M.: Meta optimization: Improving compiler heuristics with machine learning. In: Proceedings of

the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. pp. 77–90. PLDI '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/781131.781141, https://doi.org/10.1145/781131.781141

45. Stock, K., Pouchet, L.N., Sadayappan, P.: Using machine learning to improve automatic vectorization. ACM Transactions on Architecture and Code Optimization (TACO) **8**(4), 50 (2012)
46. Stratton, J.A., Rodrigues, C., Sung, I.J., Obeid, N., Chang, L.W., Anssari, N., Liu, G.D., Hwu, W.m.W.: Parboil: A revised benchmark suite for scientific and commercial throughput computing. Center for Reliable and High-Performance Computing **127** (2012)
47. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: 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. pp. 177–187. PLDI '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1542476.1542496, http://doi.acm.org/10.1145/1542476.1542496
48. Wang, Z., O'Boyle, M.F.: 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. pp. 75–84. PPoPP '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1504176.1504189, http://doi.acm.org/10.1145/1504176.1504189
49. Watkinson, N., Shivam, A., Chen, Z., Veidenbaum, A., Nicolau, A., Gong, Z.: Using hardware counters to predict vectorization. In: Rauchwerger, L. (ed.) Languages and Compilers for Parallel Computing. pp. 3–16. Springer International Publishing, Cham (2019)
50. Weiss, G.M., Provost, F.: The effect of class distribution on classifier learning: an empirical study (2001)
51. Weiss, G.M., Provost, F.: Learning when training data are costly: The effect of class distribution on tree induction. Journal of artificial intelligence research **19**, 315–354 (2003)
52. Wolfe, M.J.: High Performance Compilers for Parallel Computing. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
53. Xu, R., Tian, X., Chandrasekaran, S., Yan, Y., Chapman, B.: Nas parallel benchmarks for gpgpus using a directive-based programming model. In: Brodman, J., Tu, P. (eds.) Languages and Compilers for Parallel Computing. pp. 67–81. Springer International Publishing, Cham (2015)