CodeSeer: Input-dependent Code Variants Selection Via Machine Learning

Tao Wang
Dept. of Computer Science
North Carolina State University
twang15@ncsu.edu

Nikhil Jain
Nvidia Corporation
nikhijain@nvidia.com

David Boehme
Lawrence Livermore National Laboratory
boehme3@llnl.gov

David Beckingsale
Lawrence Livermore National Laboratory
beckingsale1@llnl.gov

Frank Mueller
Dept. of Computer Science
North Carolina State University
fmuelle@ncsu.edu

Todd Gamblin
Lawrence Livermore National Laboratory
gamblin2@llnl.gov

ABSTRACT

In high performance computing (HPC), scientific simulation codes are executed repeatedly with different inputs. The peak performance of these programs heavily depends on various compiler optimizations, which are often selected agnostically on program input or may be selected with sensitivity to just a single input. When subsequently executed, often with different inputs, performance may suffer for all or all but the one input tested, and for the latter potentially even compared to the *O3* baseline.

This work proposes a new auto-tuning framework, CodeSeer, to assess and improve existing input-agnostic or single-input centric rigid application tuning methods. Aided by CodeSeer, it is observed that modern HPC programs expose different types of input sensitivities, which present a significant challenge for prior work. To tackle this problem, CodeSeer proceeds with several machine learning models to predict the best per-input code variant on-the-fly. Our evaluation shows that CodeSeer incurs less than 0.01 second overhead, predicts the best code variant with a geometric mean precision 92% of the time and is capable of improving per-input peak performance to unprecedented levels.

CCS CONCEPTS

• **Software and its engineering → Incremental compilers**; • **Computing methodologies → Massively parallel and high-performance simulations**.

KEYWORDS

Iterative compilation, Auto-tuning, HPC, OpenMP, Code variant, NP-complete, Machine-learning

ICS '20, June 29-July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392741

ACM Reference Format:
Tao Wang, Nikhil Jain, David Boehme, David Beckingsale, Frank Mueller, and Todd Gamblin. 2020. CodeSeer: Input-dependent Code Variants Selection Via Machine Learning. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3392717.3392741

1 INTRODUCTION

In high performance computing (HPC) environments, scientists tend to test their hypotheses by running large-scale simulation codes repeatedly with different inputs. These applications occupies computing resources for a significant amount of time consuming an equally significant amount of power time and again when executed. Even a 1% performance improvement could translate into significant energy savings in such HPC production runs [16]. To extract program peak performance, various compiler-based auto-tuning technologies have been developed. These techniques are either search-based [3, 15, 33, 34, 39, 43, 45] or machine learning-based [4, 7, 18, 31]. They generally are constrained to tuning with a single input, which produces a single optimized executable evaluated on a small set of test inputs to report any performance improvements. As a result, their effectiveness may be limited by the tuning input and any performance tends to be localized to this input instead of generalizing to a larger set of inputs in production.

If an improper input is selected as the tuning input, these optimization techniques may fail to result in executables realizing any performance improvements whatsoever. Consider Fig. 1 with the relative performance (y-axis) normalized to O3 over 1,000 different code variants for Kripke [21], a discrete-ordinates solver (see Section 3). The results indicate that auto-tuning Kripke with per-program random search [43] generates 1,000 code variants with different compiler flags for Intel C++ compiler. Yet, none of them achieve any performance improvement over the O3 baseline with the selected tuning input. We call this **type-I input sensitivity (tuning sensitivity):** *The performance benefit of auto-tuning may depend on the tuning input.* **Type-I sensitivity suggests** that the tuning input has to be selected carefully, but prior work tends make selections ad-hoc. In contrast, this work CodeSeer (see Fig.2) begins with many tuning inputs, allows multiple functionally equivalent code variants, and thus eliminates the concerns of type-I sensitivity completely.

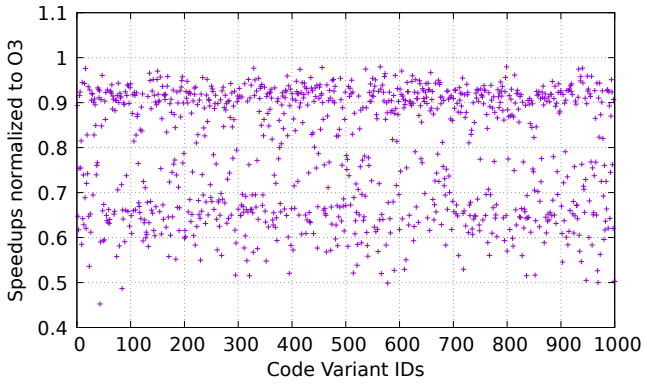


Fig. 1: Normalized speedups of 1000 Kripke code variants for a singular, identical tuning input: No performance improvement for any under O3.

Fig. 3 shows the normalized speedups for two code variants of Kripke across 1,000 inputs. Code variant 1 (cv1) and code variant 2 (cv2) are generated by choosing different compiler flags of the Intel C++ compiler. The results indicate ① **type-II input sensitivity:** *The significance of a code variant's performance for different inputs can vastly differs.* For instance, cv1 achieves a 49% speedup for input 1,000 but causes a ≈20% slowdown for input 1. **Type-II sensitivity suggests** that we should not rely on a small number (1-5) of test inputs to assess and report performance improvement for auto-tuning in practice. ② **type-III input sensitivity:** *The best performing code variants may differ for diverse test inputs.* For example, between cv1 and cv2, neither outperforms the other for all inputs. In fact, there are many cases where one performs more than 10% better than the other, e.g., cv1 is 45.7% better than cv2 for input 998, while cv2 is 28.4% better than cv1 for input 2. **Type-III input sensitivity suggests** that different code variants are necessary to realize the best per-input program performance. If a single code variant performs the best for just a subset of inputs, as shown in Fig. 3, then the optimized executable is sub-optimal and some performance is left on table. To account for type-III input sensitivity, modern optimizing compilers (e.g., Intel's compilers) feature built-in profile-guided optimization (PGO) [17]. PGO utilizes program runtime profiling information to generate optimized code for multiple characteristic inputs. However, a recent study with FuncyTuner [43] shows that PGO of Intel compilers rarely improves the performance of a set of modern scientific programs. In addition, there is little practical guidance on how to select characteristic tuning inputs for PGO.

To summarize, type-I, type-II and type-III sensitivities expose different inherent limitations of prior auto-tuning techniques that yield only one executable with one tuning input. Type-I and type-II show up at tuning and testing stages of prior techniques, respectively, while type-III challenges their fundamental assumption that only one program-level code variant is allowed. They also open up different opportunities for further improvements (see respective suggestions above). Their relationship is further discussed in Sec. 4.4.

This work proposes an auto-tuning framework, CodeSeer, to evaluate the three types of program input sensitivities, to auto-tune with many tuning/testing inputs targeting type-I/type-II sensitivities, and to perform whole-program multi-versioning for modern HPC applications targeting type-III sensitivity, unlike any prior function/loop-level multi-versioning [2, 25, 29, 30, 40] or other similar techniques in traditional static/dynamic compilers. The key insight of CodeSeer is that storage on modern supercomputers is abundant and can be traded off for program performance with fast and accurate machine learning-based prediction models to select the best code variant on a per-input basis. In particular, we formulate type-III sensitivity as a classification problem. Given an unseen input, CodeSeer classifies it as one of several representative classes and chooses to execute the corresponding code variant. As shown in Fig. 2, CodeSeer proceeds with following steps:

① It defines the training input datasets I (of size N) of a target program and prepares a set of code variants C (of size M) by randomly sampling the compiler flag space [43]. N and M should be large to represent sufficient diversity in inputs and code variants.

② It measures end-to-end runtimes for code variants C on inputs I to evaluate type-III sensitivity. If the same code variant performs the best on all inputs, it will not be further processed due to absence of type-III sensitivity; otherwise, it continues with the next step. The overhead to evaluate $M * N$ code variants and input combinations can be excessively high. Overhead reduction techniques are critical, as discussed later.

③ It selects several representative code variants according to certain criteria (see Sec. 2.5) and uses them as class labels. For each training input i, a label c is assigned such that the corresponding code variant performs the best for i.

④ It builds the classification models of CodeSeer with labeled training inputs from ③. These models predict the best code variant for unseen test inputs.

⑤ It evaluates the accuracy and overhead of multiple prediction models from ④.

To the best of our knowledge, this work is the first to investigate the three types of input sensitivities for modern HPC codes and to propose an ensemble of practical techniques to transparently improve their performance in an input-sensitive manner at whole-program granularity. In particular, the work makes the following contributions:

- Three types of input sensitivities are evaluated for a set of modern scientific applications, and a case study for Kripke [21] shows that programs with type-III sensitivity need novel tuning techniques to realize best performance.
- A solution to address type-III sensitivity is formulated as a classification problem. This problem of optimally selecting an arbitrary number of code variants to generate the training dataset (called input coverage plan generation) is proved to be NP-complete.
- A prototype machine learning-based framework, CodeSeer, to predict the per-input best code variants for programs with type-III sensitivity is created, and an ensemble of overhead reduction techniques is developed.
- CodeSeer's models are demonstrated to be effective and efficient in predicting the per-input best code variants for the set of targeted modern scientific applications.

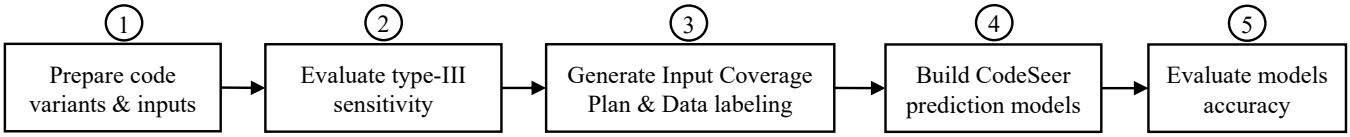


Fig. 2: Workflow of CodeSeer

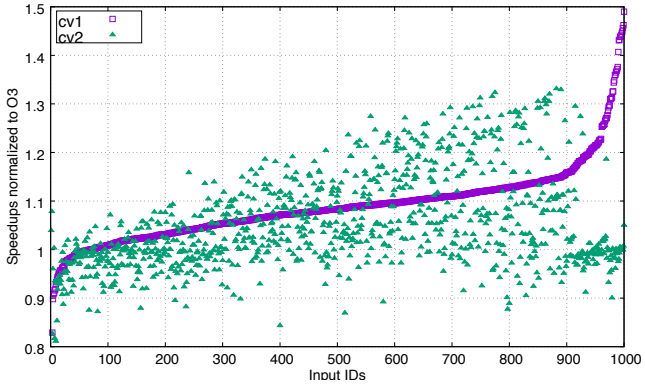


Fig. 3: Normalized speedups of two Kripke code variants across 1,000 inputs show ① type-II input sensitivity: The performance benefit of the same code variant is sensitive to different inputs; and ② type-III input sensitivity: The best performing code variants depend on sets of many inputs.

2 TACKLING TYPE-III SENSITIVITY

2.1 Problem Definition

CodeSeer addresses type-I and type-II sensitivities by using a large number of tuning inputs and testing inputs, respectively. Given a program P with type-III sensitivity, the evaluation of M code variants on an input dataset I (of size N) produces an $M \times N$ matrix ($T_{M\times N}$) of speedups relative to O3. Each row number of T identifies one code variant while each column number identifies one tuning input. The Input Coverage Plan in step ③ of Fig. 2) is to select R ($1 \le R \le M$) distinct rows that correspond to a set of R code variants (called "P's representative code variants", RC for short). Let us denote the best performing code variant in RC as c_l ($1 \le l \le R$) for an input i_n. Then, we assign the code label l to i_n (Data labeling in step ③ of Fig. 2). By labeling each input i in I, we obtain a training data set to build a model that maps (classifies) any unseen input i_t ($i_t \notin I$) to a code label l_t. The code variant corresponding to the label l_t is chosen to execute P on i_t.

2.2 Challenges

To build the classification model, the following challenges arise in preparing the labeled training dataset. First, RC has to be derived with a sufficiently large set (size M) of code variants and N inputs. If an average program run takes 10 seconds, with $M = 1000$, $N = 1000$ and 5 consecutive runs, the evaluation would take about 1.6-year machine time per program. Such high overhead must be addressed to make it practical (see Sec. 3.1 and Sec. 3.2). Second, since a smaller R implies less storage overhead (fewer code variants) and simpler prediction models that are generally more accurate with the same amount of training data, we are motivated to minimize R for a given T while maximizing the geometric mean speedups across all N inputs. We formalize and prove that the selection of an optimal RC is at least as hard as NP-complete (shown in Sec. 2.3). For computational feasibility, we design an efficient algorithm to approximate the optimal solution (see sec. 2.4).

2.3 Proof of Complexity

$$\begin{array}{c} \\ v_1 \\ v_2 \\ \dots \\ v_{|V|} \end{array} \begin{array}{c} \begin{array}{cccccccc} e_1 & e_2 & \dots & e_{|E|} & e_{1+|E|} & e_{2+|E|} & \dots & e_{|V|+|E|} \end{array} \\ \begin{pmatrix} \theta_{11} & \theta_{12} & \dots & \theta_{1|E|} & |E|-\phi_1 & 0 & 0 & 0 \\ \theta_{21} & \theta_{22} & \dots & \theta_{2|E|} & 0 & |E|-\phi_2 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \theta_{|V|1} & \theta_{|V|2} & \dots & \theta_{|V||E|} & 0 & 0 & 0 & |E|-\phi_{|V|} \end{pmatrix} \end{array}$$

Fig. 4: The Special matrix T_G for a graph G

Formally, given a $M \times N$ ($1 < M, N$) matrix T of non-negative numbers, let $S^R = \{r_1, r_2, ..., r_R\}$ be a set of distinct row numbers of T. Given T and S^R, a new matrix T' is constructed by copying each row r_i ($r_i \in S^R$, $1 \le i \le R$) of T into row i of T' once. T' is a $R \times N$ sub-matrix of T and is called the **row-set S^R-projected matrix** of T. When necessary, T' is denoted as $T(S^R)$ to stress how it is constructed. Furthermore, the **row-sum of a matrix T** is defined as $P(T) = \sum_{j=1}^{N} \max_{1\le i\le M} T_{i,j}$. Then, $P(T') = P(T(S^R)) = \sum_{j=1}^{N} \max_{1\le i\le R} T_{r_i,j}$. In total, there are $C_M^R = \binom{R}{M}$ ways to construct S^R and T' for a given R, each denoted as S_i^R ($1 \le i \le C_M^R$). We want to minimize R given by the following objective function (1):

$$\underset{R}{\text{argmin}}(\underset{S_i^R}{\text{argmax}}\, P(T(S_i^R))), \texttt{for} 1 \le R \le M, 1 \le i \le C_M^R \quad (1)$$

Let $S_\#^R = \underset{S_i^R}{\text{argmax}}\, P(T(S_i^R))$. Then, minimizing R in (1) is straightforward if the optimization for row-set $S_\#^R$ can be solved for a given R. It can be shown that the graph independent set problem [20] can be polynomially reduced to a special case of this row-sum optimization problem's decision form: Given a matrix T with R rows satisfying the aforementioned constraints, is there a row-set S^R such that $P(T(S^R))$ is at least as large as an arbitrary (but given) number C?

Given a graph $G = (V, E)$ with vertices V and edges E, and a constant k, a k-independent set is a set V^k of k vertices $\{v_{n_1}, v_{n_2}, ..., v_{n_k}\}$ s.t. $\forall 1 \le i, j \le k \wedge i \ne j$, $(v_{n_i}, v_{n_j}) \notin E$. Let us construct a special matrix T_G (see Fig. 4) with elements $\theta_{j,i}$ that has one row for each vertex, in total $|V|$ rows, and $|V| + |E|$ columns. Each of the first $|E|$ columns corresponds to one edge $e_i = (v_s, v_t)$, where $1 \le i \le |E| \wedge s \ne t \wedge 1 \le s, t \le |V|$, s.t. elements $\theta_{s,i} = \theta_{t,i} = 1 \wedge \theta_{s,s+|E|} = |E| - \sum_{i=1}^{|E|} \theta_{s,i} \wedge \theta_{j,i} = 0$ for $j \notin \{s, t\}$. As a

Algorithm 1: CodeSeer Greedy Input Coverage Plan and Data Labeling

Input : matrix $T_{M\times N}$ of M code variants for N inputs, R = # selected code variants
Output: row set S^R, labels L_N

1 $counter[1...M] = 0, L[1...N] = -1, S[1...R] = \{\}$
2 **for** $i = 1; i <= N; i{+}{+}$ **do**
3 $\quad k = \text{argmax}_k \{T[k][i] \mid 1 \le k \le M\}$
4 $\quad counter[k]{+}{+}$
5 //generate coverage plan in row-set S^R
6 **for** $i = 1; i <= R; i{+}{+}$ **do**
7 $\quad k = \text{argmax}_k \{counter[k] \mid 1 \le k \le M\}$
8 $\quad S[i] = k, counter[k] = 0$
9 //data labeling in L_N
10 **for** $i = 1; i <= N; i{+}{+}$ **do**
11 $\quad L[i] = \text{argmin}_k \{T[k][i] \mid k \in S^R = S[1...R]\}$
12 return S^R, L_N

result, the summation of all elements in any row of T_G must be equal to $|E|$ **(property p_1)**. For a k-row set S^k, let the number of ones in the same column c be m_c. Further, $\forall 1 \le c \le |E|\ m_c \in \{0, 1, 2\}$. We can prove that $P(T_G(S^k)) = k|E| \iff \forall_{1\le i\le |E|}\ m_i \le 1$ **(property p_2)**.

Proof: (1) $\Longleftarrow$: By p_1 and definition of $P(T_G(S^k))$, $P(T_G(S^k)) = k|E|$. (2) $\Longrightarrow$ by contradiction: Given $P(T_G(S^k)) = k|E|$, assume $\exists_i m_i = 2$. By definition of $P(T_G(S^k))$, for column i, only one of the two ones contributes to $P(T_G(S^k))$ and thus $P(T_G(S^k)) < k|E|$, which contradicts $P(T_G(S^k)) = k|E|$. Therefore, the assumption is false. Therefore, p_2 holds. □

A solution of k-row row-sum with $P(T_G(S^k)) = k|E|$ is also a solution for G's k-vertex independent set by p_2. Hence, we reduce the NP-complete independent set problem to a special case of the row-sum problem via the construction of T_G. Furthermore, one can verify if a given solution is at least as large as any given number C in $O(k \times N)$ (polynomial) steps, so the row-sum problem in its decision form is also in NP. Therefore, it is at least an NP-complete problem.

To summarize, a special case of the decision form of our row-sum optimization problem is NP-complete. Hence, the original problem must be at least as hard. For all practical purposes, the objective is to find an efficient algorithm to solve this problem.

2.4 Algorithm for Input Coverage Plan

For all $1 \le R \le M$, the optimal solution $S^R_\#$ subject to Equation (1) can be obtained by brute-force (BF) enumeration with time complexity $O(R \times N \times C^R_M)$, since there are C^R_M ways to select R rows to construct T', and each T' needs $R - 1$ comparisons to find N column-wise maximum elements with $O(N)$ time complexity to calculate their sum. For CodeSeer to generate an input coverage plan with objective function (1) that retains a minimal number of code variants while achieving maximal performance benefits, we prefer to begin with a small R but large N and M values. In particular, a large M represents more diversity of code variants while a large N means these code variants are evaluated thoroughly; a small R indicates fewer class labels for CodeSeer models and less difficulty to predict. Therefore, $M, N \gg R$ holds for our purpose, where brute force may be very inefficient. This motivates us to design a greedy approximation algorithm (see lines 1-8 Algorithm 1) for generating a fast input coverage plan. The intuition is to select code variants that achieve the best performance on the most inputs (largest coverage) first. The time complexity of such a selection is $O(M \times N)$ since counting the number of inputs for which a code variant achieves the best performance is iterated over the entire matrix $T_{M\times N}$ once.

2.5 Data Labeling

Since we are using classification models to predict the best per-input code variants, a labeled dataset is necessary. Note that in Algorithm 1, each row number in S^R is an identifier of a code variant in RC. Therefore, after S^R is calculated for T, data labeling (lines 9 − 11 in Algorithm 1) is performed such that each of the N inputs is labeled with the identifier of its corresponding best code variant.

3 EXPERIMENTAL DESIGN

3.1 Benchmarks and Code Variants

Our benchmark applications are ten modern HPC codes for scientific simulation (see Table 1): Kripke [21], LULESH [19], Cloverleaf [41], Optewe [38], miniFE [42], 351.bwaves, 357.bt331, 360.ilbdc, 363.swim and 370.mgrid331. The latter five applications are from the SPEC OMP 2012 suite [28]. These benchmarks are written in different languages, exploiting multi-core parallelism of modern HPC architectures via OpenMP pragmas. They are designed to resemble the properties of large-scale applications and are widely used in program performance/energy studies.

Tab. 1: List of benchmarks (LOC: lines of source code)

Name	Language	LOC	Domain
Kripke	C++	7.2k	Discrete-ordinates solver
Optewe	C++	2.7k	Seismic wave simulation
LULESH	C++	7.2k	Hydrodynamics
Cloverleaf (CL)	C, Fortran	14.5k	Hydrodynamics
miniFE	C++	24.5k	Finite Element
351.bwaves	Fortran	1.2k	Computational fluid dynamics
357.bt331	Fortran	5.4k	Computational fluid dynamics
360.ilbdc	Fortran	1.3k	Lattice Boltzmann
363.swim	Fortran	0.5k	Weather prediction
370.mgrid331	Fortran	1.7k	multigrid solver

Code variants are generated with different compilation flags of Intel compiler tool chain (version 17.0.4) to achieve high performance. As suggested by Intel [9], the original build systems are modified to use Intel's compilers, linker (xild) and archiver (xiar). The compilation flags are selected and combined with a classic random algorithm, similar to prior work [8, 43]. We also enforce strict floating-point reproducibility by always including *-fp-model source* in the compilation flags. The flags for baseline code variants are *-O3 -qopenmp -fp-model source*. End-to-end runtimes are measured on a HPC cluster with 16 computation nodes featuring two Intel Broadwell Xeon E5-2620 v4@2.1GHz processor with 16 cores (hyper-threading on) each in a two-node NUMA configuration of 64GB DRAM, running CentOS Linux release 7.5.1804. OpenMP thread placement is set to *granularity=fine,proclist=[0-15],explicit*.

We used 100 code variants in all experiments to make the runtime overhead for our benchmarks feasible, unless otherwise specified. Fig. 5 shows that the normalized speedup distributions for 1000 (top) and 100 (bottom) Kripke code variants are similar. This indicates that using 100 code variants may be sufficient to capture most of

the performance diversities for a given input (as similar patterns are seen for many other inputs for Kripke). Prior work [1, 4] also suggests that the best performance of code variants for any specific tuning input is usually found within tens to hundreds of iterations by random search within compilation flag space.

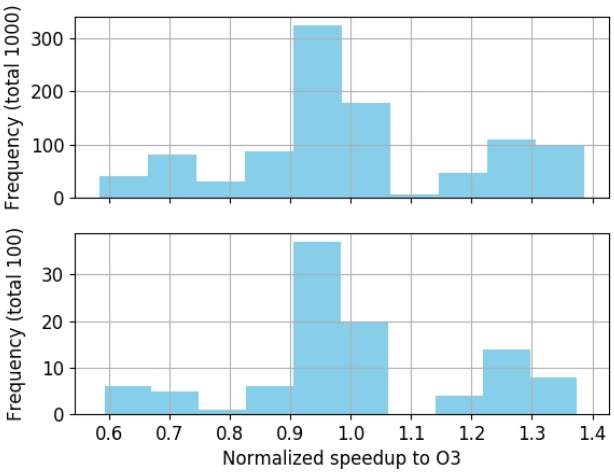


Fig. 5: Normalized speedup distributions for 1000 (top) and 100 (bottom) code variants on a single Kripke input indicating similar performance scopes: code variants in excess of 100 contribute little in extra diversity.

3.2 Prepare Input Data Sets

Inputs for these benchmarks can be set either input files or command line options. To generate a large set of inputs, we consulted the respective user guide of programs to select tunable input variables, since automatic input generation techniques [11, 35] known to us either target code coverage or only generate data arrays not appropriate for our purpose. Input variables tend to be continuous integer value ranges expressed as *variable_name* [*min_value*, *max_value*, *step_size*] or [formula (constraint)]. Variables and their ranges are independent of one another unless otherwise mentioned. Kripke [21] inputs are *groups* [2^k ($0 \le k \le 6$)], *gset* [2^k ($0 \le k \le 5$)], *legendre* [0, 9, 1], *quad* [$32k$ ($1 \le k \le 6$)], *dset* [$8k$ ($1 \le k \le 8$)], all three sub-dimensions of *zones* [$16k$ ($1 \le k \le 4$)], all three sub-dimensions *zset* [2^k ($0 \le k \le 8$)], *nest* (*GZD*). Moreover, *groups* and *quad* should divide *gset* and *dset*, respectively. LULESH [19] inputs are *size* (30, 420, 30), *i* (1, 100, 10), *r* (5, 50, 5), *c* (5, 60, 5), *b* (0, 9, 1). Cloverleaf [41] inputs are *x* and *y* (100, 18000, 200), *end_step* (10, 100, 10) Optewe [38] inputs are *x, y* and *z* (32, 1000, 32), *nt* (1, 40, 3), *s* (1, 3, 1). miniFE [42] inputs are *nx, ny, nz* (64, 328, 4). 351.bwaves inputs are *nx, ny, nz* (10, 270, 10), *method* (0, 1, 1), *configuration* (0, 1, 1), *time steps* (5, 40, 5). 357.bt331 inputs are *x, y, z* (64, 408, 4), *steps* (2, 20, 2). 360.ilbdc inputs are *x, y, z* (50, 550, 5), *GEONME* is *#channel* or *#packing* or *#synpor*. 363.swim inputs are $line_1$ (2, 20, 2), $line_6$ (30, 1000, 50), $line_8$, $line_9$ (512, 7701, 256). 370.mgrid331 inputs are *x, y, z* (64, 1200, 14), *steps* (10, 100, 10).

The input space created by ranges of all input variables are randomly sampled for 10^4 inputs without replacement. These inputs are first evaluated with the target program's O3 baseline compilation for 5 consecutive runs. Then, 4,000 of them are selected according to two criteria: (1) The end-to-end runtimes need to be within one to 200 seconds. Inputs with too short a runtime do not contribute significantly to performance (geometric mean) and are prone to measurement noise, while those with too long runtimes are prohibitively expensive for the experiments due to the number of samples taken, i.e., runs executed. (2) To reduce the overhead for evaluation, we restrict program inputs to those with less than 1% variations under O3 so that one run is sufficiently reliable to evaluate each code variant on these inputs.

The pre-selected 4,000 inputs are divided into three subsets, D_1, D_2, and D_3, by random sampling without replacement. They are used for different steps in Fig. 2. D_1 includes 1,000 inputs and is used in step ② to evaluate program input sensitivity. The evaluation is done on 100 program code variants. If a program exposes type-III sensitivity, CodeSeer uses D_1 to generate an input coverage plan that identifies a subset of 100 code variants (denoted as CV_T) as the prediction targets for data labeling in step ③. Another 2,800 inputs are evaluated on CV_T to generate more labeled inputs. Therefore, D_2 includes D_1 and these 2800 inputs. After data labeling, D_2 is used as the training dataset in step ④ to build six machine learning models. D_3 contains the remaining 200 inputs and represents the testing dataset in ⑤ for evaluation.

3.3 Build Machine Learning Models

For each program that has type-III sensitivity, we prototype six classification models in CodeSeer with Scikit-learn [36] since this model collection covers different trade-offs between prediction precision, training overhead, and online prediction speed. These models include support vector machine (SVM), decision tree (DT), random forest (RF), logistic regression (LR), neural network (NN) and K-nearest neighbors (KNN). For a target program, we first applied feature scaling to each dimension x of the training inputs (D_2) by $(x-u)/s$, where u is the mean and s is the standard deviation of x. We then employ 5-fold cross-validation with D_2 for each model to avoid over-fitting and use randomized grid search to tune their hyper-parameters with weighted F1-score as the guiding performance metric. We evaluate these models on D_3 and report their weighted precision.

4 RESULTS ON INPUT SENSITIVITY

4.1 Type-I Sensitivity

For any input i ($1 \le i \le 1000$) in D_1, there is one code variant CV^i_{opt} ($1 \le opt \le 100$) that achieves the best speedup relative to the *O3* baseline. We calculated the geometric mean speedups (S_i) over *O3* for CV^i_{opt} across all inputs in D_1 and show their probability densities in Fig. 6. We make the following observations:

(1) All applications expose type-I sensitivity since the speedups differ significantly for input i in D_1. Since CV^i_{opt} would be the final optimized executable for any traditional approach [3, 4, 32, 43], Fig. 6 demonstrates that their tuning input must be chosen carefully. If a bad input is chosen, they will not be able to find any performance improvement.

(2) Applications have very different type-I sensitivity. For example, all inputs for 357.bt always lead to performance improvement while any other benchmark requires more effort to identify the best tuning input. In contrast, CodeSeer considers input sensitivity in

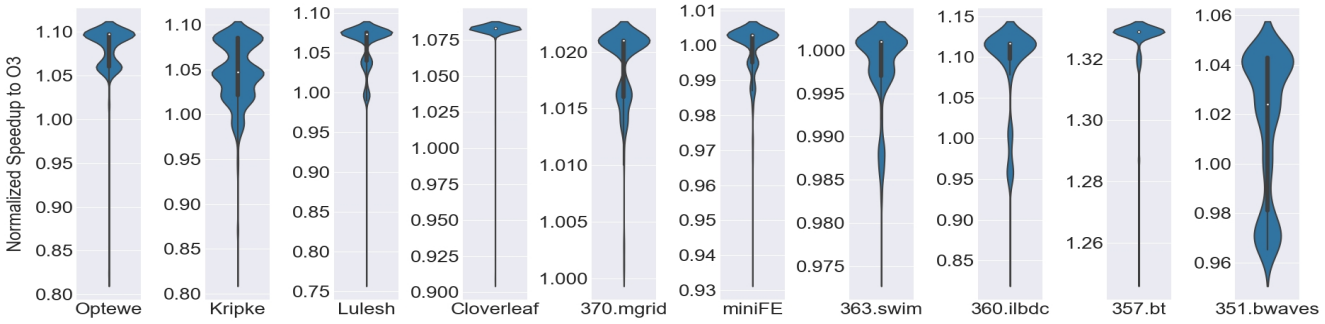


Fig. 6: All benchmarks expose type-I sensitivity: Performance benefits of a tuned program depends on the tuning input used.

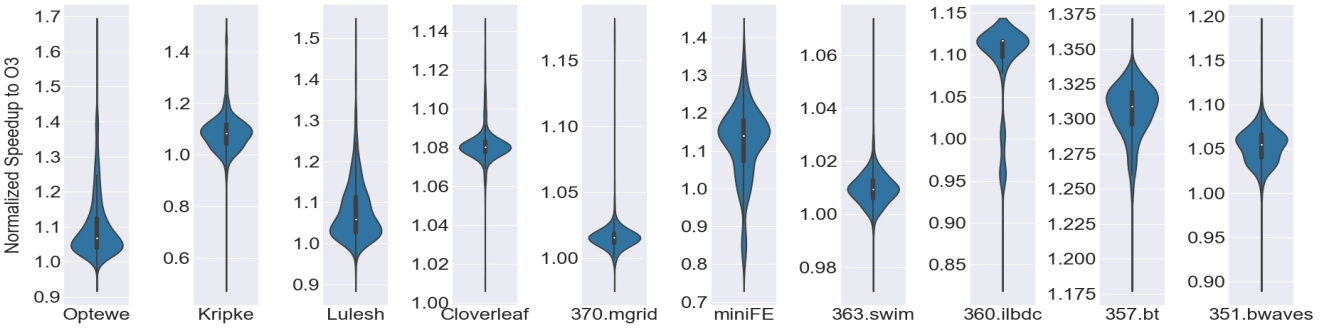


Fig. 7: All benchmarks expose type-II sensitivity: Performance benefits are different for inputs in D_1 with CV_{opt} that achieves the best geometric mean speedup on D_1.

its design by evaluating more tuning inputs, thereby avoiding a single-input bias.

4.2 Type-II Sensitivity

For a target program, let CV_{opt} be the code variant that achieves the highest geometric mean speedups across all inputs in D_1. The distribution of speedups achieved by CV_{opt} for inputs in D_1 is shown in Fig. 7. We observe that all applications expose type-II sensitivity: CV_{opt} achieves very different speedups for inputs in D_1. Prior work [3, 4, 32, 43] evaluated performance of optimized executables on a limited number of testing inputs resulting in performance benefits that may not generalize to other inputs. In the worst case, their optimized executables may lead to degradation, sometimes as significant as miniFE and 360.ilbdc with a minimum of 0.72 and 0.82, respectively. CodeSeer avoids such limitations by evaluating many more (200) testing inputs.

4.3 Type-III Sensitivity

The cumulative number of inputs with different speedup gaps between the CV_{opt} and the best code variant (CV^i_{opt}) for input i are shown in Fig. 8. We observe that CV_{opt} is not always the per-input best code variant (type-III sensitivity): For Cloverleaf, CV_{opt} *is* the best per-input code variant, but Kripke under CV_{opt} only achieves the best per-input speedups for about 52% of the inputs in D_1. All other applications fall in between these two extremes. In fact, CV_{opt} for Optewe, 370.mg, miniFE, 363.swim and 357.bt differs by less than 2% (performance loss) from CV^i_{opt} for around 8%, 1%, 10%, 1%, 3% and 5% of the inputs in D_1, respectively; CV_{opt} for Lulesh, 351.bwaves and 360.ilbdc is over 3% inferior to more than 10% inputs, respectively.

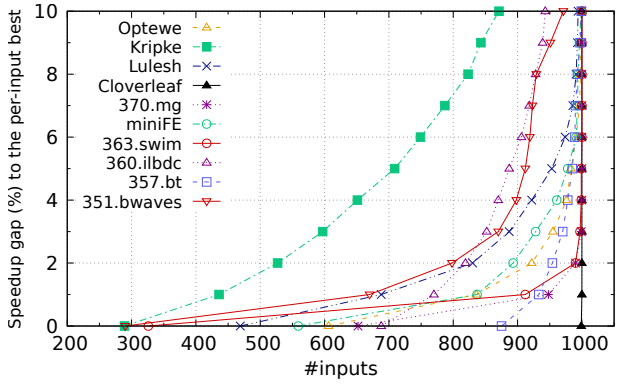


Fig. 8: Cumulative number of inputs in D_1 with different normalized speedup gaps between CV_{opt} and CV^i_{opt}

Results for Kripke (Fig. 3), 351.bwaves, 360.ilbdc and Lulesh (Fig. 9) show that there are two code variants that perform best for different subsets of D_1. These two code variants happen to be identified by our greedy algorithm (see Algorithm. 1), while the

problem in general is at least NP-complete. We observe that type-III sensitivity introduces an inherent limitation for prior auto-tuning approaches that yield only one optimized executable for each target program, e.g., 10% of D_1 experience significant degradation relative to their *O3* baseline while the performance of more than 10% of inputs for Kripke CV_{opt} results in a 10% to 52.8% gap relative to peak performance. This reinforces the need for low-overhead high-precision predictive models.

4.4 Discussion on Type-III Sensitivity

The two extremes in Fig. 8 of Cloverleaf and Kripke demonstrate that **type-III sensitivity is a stronger version of type-II sensitivity** in that if type-III sensitivity holds for a program, type-II sensitivity must hold, but not vice visa. In addition, type-III sensitivity is indicative of auto-tuning opportunities to improve program per-input performance and thus deserves a separate classification.

Type-III sensitivity may result from two scenarios. First, many compiler optimizations, e.g., loop unrolling and vectorization, do not always improve performance. For this reason, they usually rely on heuristic cost models to decide whether or not to perform the optimizations at *compile time*, and/or subsequently decide whether or not to take the optimized code paths at *runtime*. However, the cost model may mispredict. For the two code variants (CV_1, CV_2) of Kripke in Fig. 3, we profiled a hot loop (LTimes) with Caliper [6] and Likwid [14] for two inputs, i_1 and i_2. We observed that:

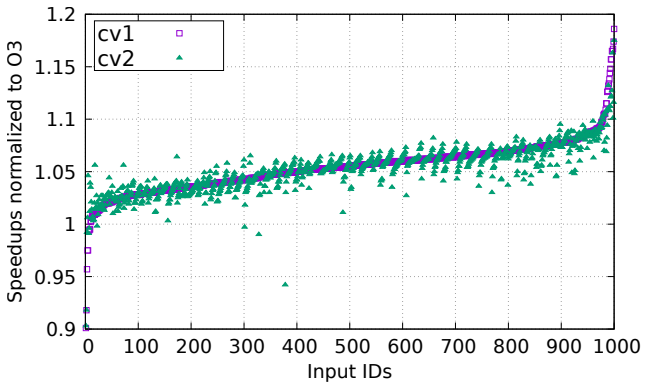
(1) CV_1 is 41% faster than CV_2 on i_1, but 32% slower on i_2;
(2) LTimes is vectorized in CV_1 but unvectorized in CV_2.

These observations demonstrate that Intel's C compiler has a cost model that fails for these inputs. Second, runtime ratios of hot loops may change due to different input sizes. Even if the relative loop speedups in different code variants stay the same, such changes may favor another code variant for a new input. In general, our experiences suggest that it is challenging to understand the performance discrepancies between code variants on a per-input basis: Differences may be subsumed by cache behavior or memory bandwidth utilization. In contrast to past work, CodeSeer utilizes machine learning models to capture the complex relationship between *program inputs and code variants* resulting from different optimizations.

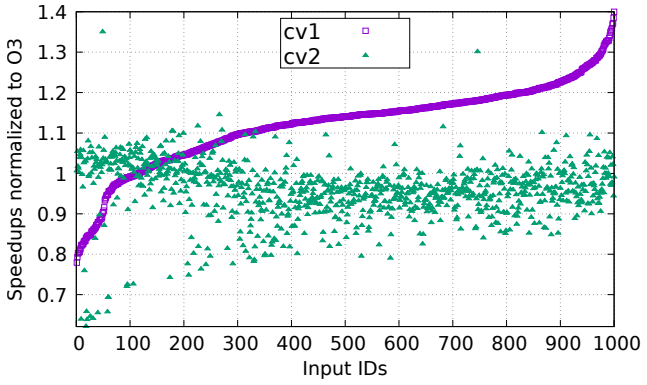
5 MODEL-BASED TUNING RESULTS

5.1 Precise Prediction

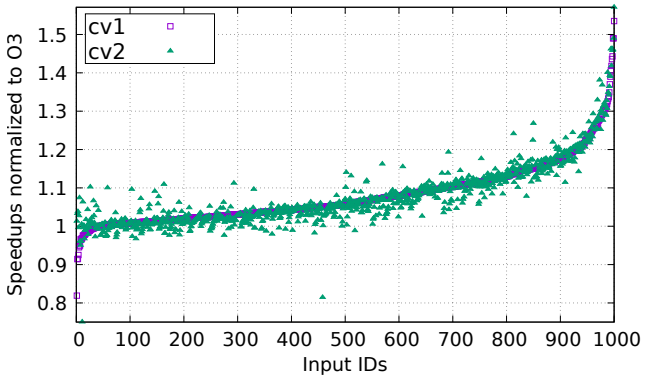
5.1.1 Deciding Target Code Variants. CodeSeer relies on its greedy algorithm (see Algorithm. 1) to identify R ($1 \leq R \leq 100$) code variants as the input coverage plan for D_1. Fig. 10 shows that the benefit of a model with more target classes, assuming perfect prediction, diminishes very fast and that a model with two code variants is able to harvest most of the performance potential. Moreover, since there is a one-to-one mapping from these R code variants to R target classes, when the training set size is fixed, having more classes means that there will be fewer samples in each class and may produce less precise models. These observations were the motivation to use only two target classes (CV_T) for CodeSeer models in our work .



(a) 351.bwaves



(b) 360.ilbdc



(c) Lulesh

Fig. 9: 360.ilbdc, 351.bwaves, and Lulesh expose type-III sensitivity: CV1 and CV2 are the two code variants selected by CodeSeer's greedy coverage plan.

5.1.2 Precision Results. Inputs in D_2 (excluding those in D_1), are evaluated on the two code variants in D_1's coverage plan. After data labeling (see Algorithm. 1), D_2 is used for training CodeSeer's models and D_3 (see 3.2) is used as the test set. D_3 approximates the class imbalance in D_2 and is an unbalanced test set. In contrast, a balanced test set has the same number of inputs in each target class, which is useful to evaluate how well CodeSeer's models can

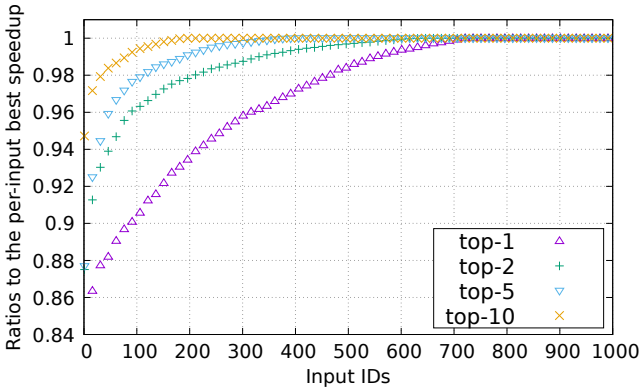
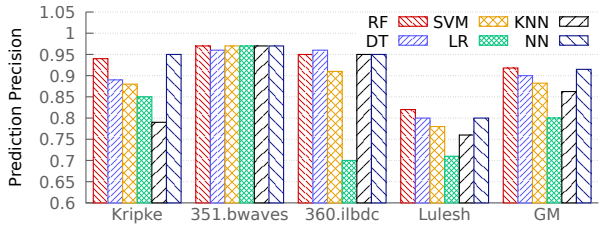
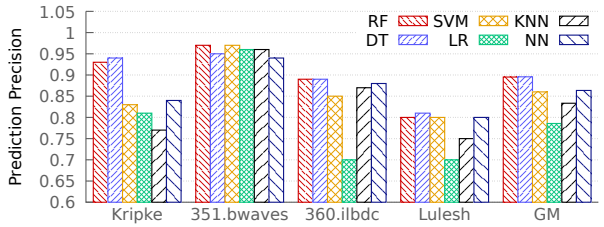


Fig. 10: Ratios (for Kripke only) to the per-input best speedup: higher is better. The per-input best speedup is the maximum speedup achieved by 100 code variants for inputs in D_1; top-K (1, 2, 5, 10) indicates the best per-input speedup achieved by K code variants identified by CodeSeer's greedy algorithm.

generalize to other datasets. To this end, we upsampled the input space for a few more inputs to enlarge the smaller class up to 100 inputs, and conversely downsampled D_3 to reduce the larger class down to 100 inputs. Fig. 11 shows CodeSeer's prediction results. We make the following observations:



(a) unbalanced test set



(b) balanced test set

Fig. 11: Prediction precision of six different models for the unbalanced (a) and balanced (b) testing datasets: RandomForest (RF) achieves 92% and 90% geometric mean precision, respectively.

(1) Random forest (RF) achieves the best geometric mean precision, 92% and 90% on the unbalanced and balanced datasets, respectively. Such high precision shows that RF is able to generalize well.

(2) Precision varies for different applications. 351.bwaves experiences 97% precision on both test sets; precisions for Kripke and

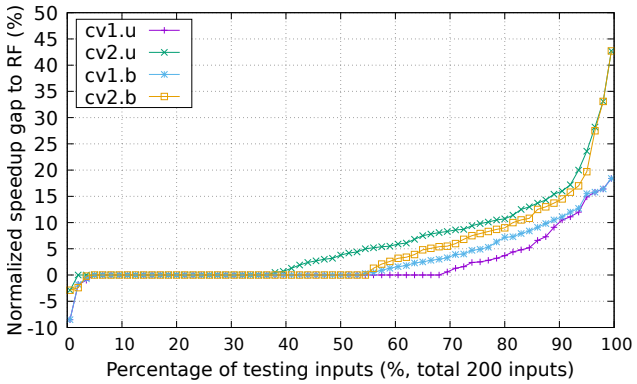


Fig. 12: Performance benefits of CodeSeer's RF model for Kripke. Figures for 351.bwaves, 360.ilbdc and Lulesh are similar (omitted due to space).

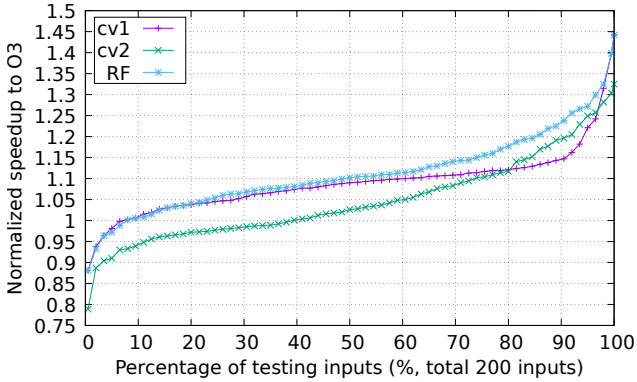


Fig. 13: Speedups normalized to O3 for Kripke CV_1, CV_2, and the RF model. The results for the balanced test set are similar (omitted).

360.ilbdc are around 95%; Lulesh's precision is about 80%, which results from mispredicting the best code variant in absolute terms, but only to choose another code variant that is only 1% less performant for these inputs, i.e., results are nearly indistinguishable. If precision was redefined to allow variations of up to 1%, prediction precision for RF would increase to 95% in the figure.

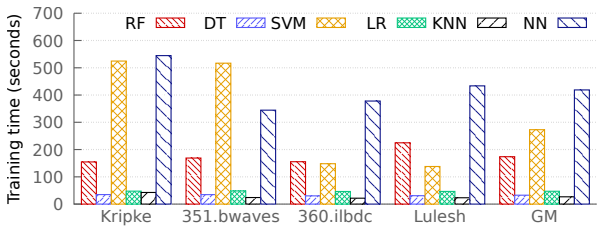


Fig. 14: CodeSeer model training overhead for all models on D_2: training RandomForest (RF) takes less than 4 minutes for all applications.

5.1.3 Performance Benefits. As mentioned in Section. 4.1, CodeSeer first handles type-I sensitivity by tuning with many more inputs

and identifies CV_{opt} with its greedy algorithm. CodeSeer's high-precision prediction model, random forest (RF), then improves the per-input performance to an unprecedented level. This is illustrated by Fig. 12, which plots the difference in performance between RF and the respective code variants (balanced/unbalanced CVs) for all inputs. RF outperforms any CV a positive gap, which is the case for almost all inputs, sometimes by as much as 43% (Speedup gaps are sorted independently in monotonically increasing order for each data set). Fig. 13 shows that the RF model provides the best result in terms of its speedup factor over O3, up to a 45% improvement with benefits for 93% of the input. In contrast, among the code variants CV_1 and CV_2, one may provide higher performance for some inputs while the inverse is the case for other inputs. Again data sets are sorted independently. For unsorted (same input) plots, scatter graphs would be similar to Fig. 1), i.e., the RF model has to compensate for complex input sensitivities. Overall, we observed that:

(1) CodeSeer's correct RF predictions translate into a performance benefit for all applications on both unbalanced and balanced test sets exceeding those of CV_{opt}. For Kripke, around 20% and 25% of the inputs benefit from more than 5% more speedups (relative to the O3 baseline) compared to CV_1 and CV_2, respectively.

(2) CodeSeer's RF mispredictions have limited negative impacts, since the rate is low (Kripke, 351.bwaves) or the speedup differences for mispredicted inputs are small (mostly within 1% for Lulesh). For 360.ilbdc, the training set is highly unbalanced against CV_2, so the misprediction rate for CV_2 is higher than CV_1 on the balanced test set.

5.2 Overhead Analysis

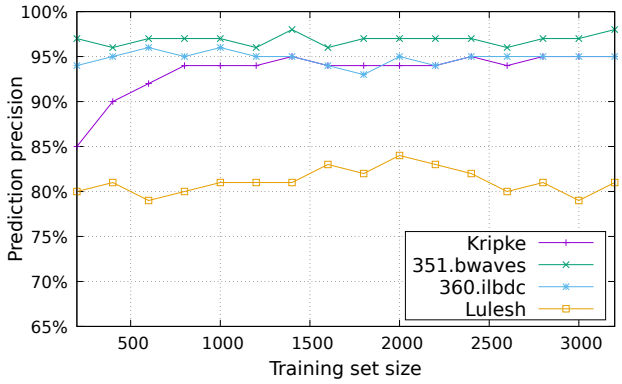


Fig. 15: RF precision converges with 1000 training inputs for an unbalanced test set of 200 samples.

On our 16 node Broadwell cluster, it takes about one week per program to evaluate their input sensitivity with several optimizations, e.g., using 100 code variants (see Sec. 3.1). Evaluation of 3,000 more inputs in CV_{opt} takes less than one day, which may be further reduced. In Fig. 15, we observed that random forest converges with about 1,000 training inputs for an unbalanced test set (with similar results for a balanced test set). CodeSeer's model training overhead, shown in Fig. 14, amounts to less than 4 minutes to train RF for each application. Since CodeSeer's model is designed to perform online prediction at the time that the target program is being launched, fast inference is critical. We observed that inference overhead is around 0.01 second for CodeSeer's RF models after being translated from a *python* implementation in Scikit-learn [36] to a *C* implementation with a model translator [37]. Such a low overhead makes CodeSeer's RF model viable for per-input code variant selection in HPC environments where programs run for hours or days. Although CodeSeer with two code variants roughly doubles the code size (several extra MBs), it is still a reasonable trade-off since a modern HPC node has several Peta-byte storage capacity.

6 RELATED WORK

The most closely related work to ours focuses on program performance with iterative compiling techniques. These approaches are either search-based [3, 8, 15, 33, 34, 39, 45] or machine learning-based [4, 7, 18, 23, 31]. FuncyTuner [43] is the state-of-art approach for optimizing modern scientific applications with caliper-guided [6] fine-grained random search. FuncyTuner optimizes multiple hot loops simultaneously, while others [15, 23, 45] focus on a single loop at a time. Machine learning-based approaches [4, 7, 18, 31] aim to reduce the tuning overhead of search-based approaches while striving to achieve similar performance. However, these techniques usually evaluate the impact of program input sensitivity either for non-HPC codes or with few inputs, which differs significantly from our work. Moreover, Yang Chen et al. [8] studied program input sensitivity to determine a single code variant for a given embedded application. They observed that a program-specific compiler flag combination can achieve most of the best performance in their experiments. However, our evaluation shows that modern HPC codes may experience type-III input sensitivity and CodeSeer's predictive models (search-based approaches are not feasible due to whole-program code variants) are critical to achieve the best per-input performance.

Prior work also focuses on mapping program inputs to the best algorithmic variants or program configurations at runtime. Jae-Seung Yeom et al. [44], Jiajia Li et al. [22], JunHong Liu et al. [24] and Yufei Ding et al. [10] used machine learning models to learn such mappings from features of matrices to characterize solvers or algorithmic kernels. Frigo [12] showed that a special-purpose compiler can generate specialized DFT kernels for a given input. These techniques are tied to a specific algorithm while CodeSeer has no assumption on the tuning target and is much more general. Other research [2, 25, 27, 29, 30, 40, 47] focuses on auto-tuning HPC kernels for modern GPUs or optimal runtime configurations [46]. For example, Magni et al. [25] propose to choose the best parameters for transforming sequential OpenACC loops into parallel threaded codes while considering program input sizes. Tillet et al. [40] build a multi-layered perceptron model to predict the performance of code variants with different tuning parameters. The model incorporates the impact of inputs and is able to guide the construction of the per-input best code variant at runtime. Muralidharan et al. [29, 30] expose tunable parameters to programmers for code variant generation and construct a machine learning model for per-input code variant selection. These works focus on single GPU kernel selection at runtime while CodeSeer tackles type-III sensitivity for programs with multiple kernels at launch-time and whole-program granularity. Zhang et al. [27, 47] propose an auto-tuner for 3D stencil

code generation on several heterogeneous GPUs. They exhaustively search a space composed of GPU thread block dimensions and memory placements but do not consider the per-input optimal code variant selection.

As we approach exascale computing systems in HPC, power efficiency becomes more desirable due to high peak power consumption [16]. Gholkar et al. [13] proposed techniques to improve HPC power efficiency at both the machine level and the job level. Bari et al. [5] developed a framework called ARCS to automatically select the best runtime configurations for each OpenMP parallel region at a given power level. Marathe et al. [26] modeled the interactions between an application's input parameters and system constraints such as power usage with deep transfer learning so that the best performing configuration can be identified quickly. Sourouri [38] enforced fine-grained dynamic voltage and frequency scaling (DVFS) and OpenMP configurations to reduce energy consumption. These techniques emphasize the importance of runtime configurations to program performance and power efficiency, which are orthogonal to CodeSeer's compilation-time approach.

7 CONCLUSION

We proposed a novel auto-tuning framework, CodeSeer, that discovers program input sensitivity for modern HPC applications and performs per-input code variant selection via fast and precise machine learning models. With CodeSeer, we identified three types of program sensitivities. We showed that (1) the optimized executable produced for a target program by prior auto-tuning techniques is sensitive to the tuning input (type-I sensitivity); (2) the performance benefit of the optimized executable reported by prior techniques may not generalize to many unseen inputs in practice (type-II sensitivity); and (3) the per-input best speedup may depend on different code variants (type-III sensitivity).

In contrast to prior auto-tuning techniques, CodeSeer address the fundamental challenges of program input sensitivity by optimizing programs for many more inputs than prior work and by generating machine learning models to automatically perform per-input code variant selection at program launch-time. Evaluations show that CodeSeer's models achieved 90% and 92% precision (geometric mean) on balanced and unbalanced testing sets, respectively, while the prediction overhead is around 0.01 seconds. CodeSeer contributes a novel, computationally viable technique for improving performance for HPC programs with type-III sensitivity to an unprecedented level.

ACKNOWLEDGMENTS

This work was funded in part by NSF grant 1525609, by US Air Force, Office of Scientific Research grant AFOSR-FA9550-17-1-0205, by subcontracts LLNL-B627261 and LLNL-B631308 from Lawrence Livermore National Laboratory, and under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 and supported by the Office of Science, Office of Advanced Scientific Computing Research as well as the Advanced Simulation and Computing (ASC) program.

REFERENCES

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. 2006. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*. 11 pp.–. https://doi.org/10.1109/CGO.2006.37

[2] M. Ahmad, H. Dogan, C. J. Michael, and O. Khan. 2019. HeteroMap: A Runtime Performance Predictor for Efficient Processing of Graph Analytics on Heterogeneous Multi-Accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 268–281. https://doi.org/10.1109/ISPASS.2019.00039

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[4] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *ACM Trans. Archit. Code Optim.* 13, 2, Article 21 (June 2016), 25 pages. https://doi.org/10.1145/2928270

[5] M. A. S. Bari, N. Chaimov, A. M. Malik, K. A. Huck, B. Chapman, A. D. Malony, and O. Sarood. 2016. ARCS: Adaptive Runtime Configuration Selection for Power-Constrained OpenMP Applications. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 461–470. https://doi.org/10.1109/CLUSTER.2016.39

[6] D. Boehme, T. Gamblin, D. Beckingsale, P. T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz. 2016. Caliper: Performance Introspection for HPC Software Stacks. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. 550–560. https://doi.org/10.1109/SC.2016.46

[7] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, Washington, DC, USA, 185–197. https://doi.org/10.1109/CGO.2007.32

[8] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. 2010. Evaluating Iterative Optimization Across 1000 Datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 448–459. https://doi.org/10.1145/1806596.1806647

[9] Intel Corporation. 2018. Step by Step Performance Optimization with Intel® C++ Compiler. https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler. (2018).

[10] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 379–390. https://doi.org/10.1145/2737924.2737969

[11] Richard Draves and Robbert van Renesse (Eds.). 2008. *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association.

[12] Matteo Frigo. 1999. A Fast Fourier Transform Compiler. *SIGPLAN Not.* 34, 5 (May 1999), 169–180. https://doi.org/10.1145/301631.301661

[13] Neha Gholkar, Frank Mueller, and Barry Rountree. 2016. Power Tuning HPC Jobs on Power-Constrained Systems. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2967938.2967961

[14] G. Hager, G. Wellein, and J. Treibig. 2010. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *2010 39th International Conference on Parallel Processing Workshops(ICPPW)*, Vol. 00. 207–216. https://doi.org/10.1109/ICPPW.2010.38

[15] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. 2010. Loop Transformation Recipes for Code Generation and Auto-Tuning. In *Languages and Compilers for Parallel Computing*, Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–64.

[16] Mark Halper. 2015. Supercomputing's Super Energy Needs, and What to Do About Them. https://cacm.acm.org/news/192296-supercomputings-super-energy-needs-and-what-to-do-about-them/fulltext. (September 2015).

[17] Intel. 2018. Profile-Guided Optimizations Overview. https://software.intel.com/en-us/node/522721. (Oct. 2018).

[18] M. R. Jantz and P. A. Kulkarni. 2013. Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 1–10. https://doi.org/10.1109/CASES.2013.6662511

[19] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. 2013. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *27th*

IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013). Boston, USA.

[20] Carl Kingsford. 2019. Reductions & NP-completeness. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/npcomplete.pdf. (June 2019).

[21] Adam Kunen, Teresa S. Bailey, and Peter N. Brown. 2015. kripke - a massively parallel transport mini-app. American Nuclear Society.

[22] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: An Input Adaptive Auto-tuner for Sparse Matrix-vector Multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 117–126. https://doi.org/10.1145/2491956.2462181

[23] Junhong Liu, Guangming Tan, Yulong Luo, Jiajia Li, Zeyao Mo, and Ninghui Sun. 2019. An Autotuning Protocol to Rapidly Build Autotuners. *ACM Trans. Parallel Comput.* 5, 2, Article Article 9 (Jan. 2019), 25 pages. https://doi.org/10.1145/3291527

[24] Yulong Luo, Guangming Tan, Zeyao Mo, and Ninghui Sun. 2015. FAST: A Fast Stencil Autotuning Framework Based On An Optimal-Solution Space Model. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. Association for Computing Machinery, New York, NY, USA, 187–196. https://doi.org/10.1145/2751205.2751214

[25] Alberto Magni, Dominik Grewe, and Nick Johnson. 2013. Input-aware Auto-tuning for Directive-based GPU Programming. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*. ACM, New York, NY, USA, 66–75. https://doi.org/10.1145/2458523.2458530

[26] Aniruddha Marathe, Rushil Anirudh, Nikhil Jain, Abhinav Bhatele, Jayaraman Thiagarajan, Bhavya Kailkhura, Jae-Seung Yeom, Barry Rountree, and Todd Gamblin. 2017. Performance Modeling Under Resource Constraints Using Deep Transfer Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 31, 12 pages. https://doi.org/10.1145/3126908.3126969

[27] Frank Mueller and Yongpeng Zhang. 2013. Hidp: A Hierarchical Data Parallel Language. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/CGO.2013.6494994

[28] Matthias S. Müller, John Baron, William C. Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, Pavel Shelepugin, Matthijs van Waveren, Brian Whitney, and Kalyan Kumaran. 2012. SPEC OMP2012 — An Application Benchmark Suite for Parallel Systems Using OpenMP. In *OpenMP in a Heterogeneous World*, Barbara M. Chapman, Federico Massaioli, Matthias S. Müller, and Marco Rorro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 223–236.

[29] Saurav Muralidharan, Amit Roy, Mary Hall, Michael Garland, and Piyush Rai. 2016. Architecture-Adaptive Code Variant Tuning. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 325–338. https://doi.org/10.1145/2872362.2872411

[30] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro. 2014. Nitro: A Framework for Adaptive Code Variant Tuning. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 501–512. https://doi.org/10.1109/IPDPS.2014.59

[31] Ricardo Nobre, Luiz G. A. Martins, and João M. P. Cardoso. 2016. A Graph-based Iterative Compiler Pass Selection and Phase Ordering Approach. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES 2016)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/2907950.2907959

[32] Zhelong Pan and R. Eigenmann. 2006. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimization (CGO'06)*. 12 pp.–. https://doi.org/10.1109/CGO.2006.38

[33] Zhelong Pan and Rudolf Eigenmann. 2008. PEAK&Mdash;a Fast and Effective Performance Tuning System via Compiler Optimization Orchestration. *ACM Trans. Program. Lang. Syst.* 30, 3, Article 17 (May 2008), 43 pages. https://doi.org/10.1145/1353445.1353451

[34] Mihail Popov, Chadi Akel, William Jalby, and Pablo de Oliveira Castro. 2016. Piecewise Holistic Autotuning of Compiler and Runtime Parameters. In *Euro-Par 2016 Parallel Processing - 22nd International Conference (Lecture Notes in Computer Science)*, Christos Kaklamanis, Theodore S. Papatheodorou, and Paul G. Spirakis (Eds.), Vol. 9833. 238–250.

[35] M. Rodrigues, B. Guimarães, and F. M. Q. Pereira. 2019. Generation of In-Bounds Inputs for Arrays in Memory-Unsafe Languages. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 136–148. https://doi.org/10.1109/CGO.2019.8661178

[36] scikit-learn community. 2019. Scikit-learn: Machine Learning in Python. https://scikit-learn.org/stable/index.html. (2019).

[37] scikit-learn community. 2019. Sklearn-porter: Transpile trained scikit-learn estimators to C, Java, JavaScript and others. https://github.com/nok/sklearn-porter. (2019).

[38] Mohammed Sourouri, Espen Birger Raknes, Nico Reissmann, Johannes Langguth, Daniel Hackenberg, Robert Schöne, and Per Gunnar Kjeldsberg. 2017. Towards Fine-grained Dynamic Tuning of HPC Applications on Modern Multi-core Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 41, 12 pages. https://doi.org/10.1145/3126908.3126945

[39] Jayaraman J. Thiagarajan, Nikhil Jain, Rushil Anirudh, Alfredo Gimenez, Rahul Sridhar, Aniruddha Marathe, Tao Wang, Murali Emani, Abhinav Bhatele, and Todd Gamblin. 2018. Bootstrapping Parameter Space Exploration for Fast Tuning. In *Proceedings of the 2018 International Conference on Supercomputing (ICS '18)*. ACM, New York, NY, USA, 385–395. https://doi.org/10.1145/3205289.3205321

[40] Philippe Tillet and David Cox. 2017. Input-aware Auto-tuning of Compute-bound HPC Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 43, 12 pages. https://doi.org/10.1145/3126908.3126939

[41] UK-MAC. 2018. Cloverleaf mini-apps. http://uk-mac.github.io/CloverLeaf/. (2018).

[42] UK-MAC. 2019. Mantevo mini-apps. https://mantevo.github.io/download.html. (2019).

[43] Tao Wang, Nikhil Jain, David Beckingsale, David Boehme, Frank Mueller, and Todd Gamblin. 2019. FuncyTuner: Auto-tuning Scientific Applications With Per-loop Compilation. In *Proceedings of the Fourty-Eighth International Conference on Parallel Processing (ICPP)*.

[44] J. Yeom, J. J. Thiagarajan, A. Bhatele, G. Bronevetsky, and T. Kolev. 2016. Data-Driven Performance Modeling of Linear Solvers for Sparse Matrices. In *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 32–42. https://doi.org/10.1109/PMBS.2016.009

[45] Qing Yi. 2012. POET: A Scripting Language for Applying Parameterized Source-to-source Program Transformations. *Softw. Pract. Exper.* 42, 6 (June 2012), 675–706. https://doi.org/10.1002/spe.1089

[46] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-Aware High Dimensional Configurations Auto-Tuning of In-Memory Cluster Computing. *SIGPLAN Not.* 53, 2 (March 2018), 564–577. https://doi.org/10.1145/3296957.3173187

[47] Yongpeng Zhang and Frank Mueller. 2012. Auto-generation and Auto-tuning of 3D Stencil Codes on GPU Clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 155–164. https://doi.org/10.1145/2259016.2259037