Learning Gradual Typing Performance

Mohammad Wahiduzzaman Khan
CACS, University of Louisiana, Lafayette, LA, USA

Sheng Chen
CACS, University of Louisiana, Lafayette, LA, USA

Yi He
Data Science, College William & Mary, Williamsburg, VA, USA

Abstract

Gradual typing has emerged as a promising typing discipline for reconciling static and dynamic typing, which have respective strengths and shortcomings. Thanks to its promises, gradual typing has gained tremendous momentum in both industry and academia. A main challenge in gradual typing is that, however, the performance of its programs can often be unpredictable, and adding or removing the type of a a single parameter may lead to wild performance swings. Many approaches have been proposed to optimize gradual typing performance, but little work has been done to aid the understanding of the performance landscape of gradual typing and navigating the migration process (which adds type annotations to make programs more static) to avert performance slowdowns.

Motivated by this situation, this work develops a machine-learning-based approach to predict the performance of each possible way of adding type annotations to a program. On top of that, many supports for program migrations could be developed, such as finding the most performant neighbor of any given configuration. Our approach gauges runtime overheads of dynamic type checks inserted by gradual typing and uses that information to train a machine learning model, which is used to predict the running time of gradual programs. We have evaluated our approach on 12 Python benchmarks for both guarded and transient semantics. For guarded semantics, our evaluation results indicate that with only 40 training instances generated from each benchmark, the predicted times for all other instances differ on average by 4% from the measured times. For transient semantics, the time difference ratio is higher but the time difference is often within 0.1 seconds.

2012 ACM Subject Classification Theory of computation → Type structures; Computing methodologies → Machine learning; Computing methodologies → Learning linear models

Keywords and phrases Gradual typing performance, type migration, performance prediction, machine learning

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.21

Supplementary Material *Software (Source Code)*: `https://github.com/wahid-nlogn/ECOOP_2024_MLGTP`, archived at `swh:1:dir:3f8fb3b4e2160fa825b6f823185bf8e2a7e1ec92`

Funding This work has been supported in part by the National Science Foundation (NSF) under Grant Nos. IIS-2245946, IIS-2236578, and CCF-1750886 and in part by the Commonwealth Cyber Initiative (CCI) and DARPA.

1 Introduction

Statically typed languages offer benefits such as early programming error detection, documentation, and better performance but can hinder program executions when they are incomplete or contain type errors. Dynamically-typed languages offer the benefits of fast prototyping and flexible usability but provide less program correctness guarantee. Traditionally, languages are either static or dynamic. In an effort to reconcile these typing disciplines, a typing discipline named *gradual typing* was developed and popularized in the last decade Siek and Taha [38],

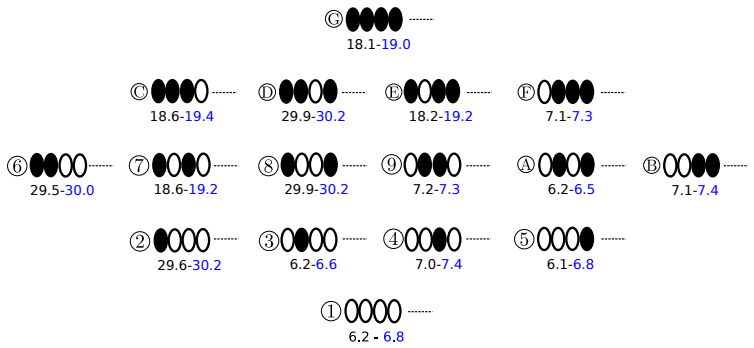

Figure 1 Part of the performance lattice for the Pascal benchmark. The lattice consists of 16 configurations, a combination of four parameters with each being typed or untyped. Each filled (unfilled) oval represents a typed (untyped) parameter. Each configuration shows only 4 ovals and omits the rest, which is the same across the 16 configurations. A circled number or letter is attached to each configuration for easy reference in the paper. Each configuration is associated with two times, separated by a '-'. The first time is the measured time of the configuration and the second (in blue) is the predicted time by our machine learning algorithm. All times are in seconds in the paper.

Siek and Vachharajani [39], Garcia and Cimini [13], Tobin-Hochstadt and Felleisen [42], Tobin-Hochstadt et al. [43], Campora et al. [6], Castagna et al. [9], Migeed and Palsberg [22], Phipps-Costin et al. [31], Greenman and Felleisen [14].

The main idea of gradual typing is that within a single program, parts of it may be statically typed (by giving type annotations to parameters in that part) and parts of it may be dynamically typed (by leaving out type annotations to parameters or explicitly giving them the dynamic type, written as `Dyn`). Ideally, in gradual typing, prototyping and initial development is done with the dynamic aspect of the language, and programs are migrated to static aspect when performance and correctness becomes critical.

The goal of type migration is to add type annotations to parameters with dynamic types of a program. A commonly used notion in type migration is *configurations* Greenman et al. [15]. For any program, a configuration specifies which subset of all the parameters are typed. For example, in the fully dynamic configuration, this subset is empty, and in the fully static configuration, this subset includes all parameters. For a program with n parameters, there can be up to 2^n configurations since each parameter can be typed or untyped. We can organize all the configurations into a lattice such that the set of typed parameters in the join of two configurations is a union of those of the two configurations. To illustrate, Figure 1 presents a part of the lattice for the Pascal benchmark in Python.

1.1 Performance Problem in Type Migration

There are two issues related to gradual type migration:

(1) finding parameters where type annotations could be added and

(2) understanding performance changes and maintaining good (acceptable) performance as type annotations are added.

For issue 1, a lot of work has been done to automatically adding type annotations to dynamically-typed programs, including static approaches Castagna et al. [9], Kristensen and Møller [19], Campora and Chen [7], Rastogi et al. [34], Chandra et al. [10], Siek and Vachharajani [39], dynamic approaches Miyazaki et al. [24], Cristiani and Thiemann [11], and machine learning based approaches Mir et al. [23], Peng et al. [30], Pradel et al. [33],

Allamanis et al. [3]. Several approaches have also been developed to find best migrations in the sense of adding type annotations to as many parameters as possible Campora et al. [6], Migeed and Palsberg [22], Phipps-Costin et al. [31]. Issue 2, however, has received less attention.

While it is tempting to integrate all the type annotations suggested by a type migration tool Castagna et al. [9], Kristensen and Møller [19], Campora and Chen [7], Rastogi et al. [34], Chandra et al. [10], Siek and Vachharajani [39], Mir et al. [23], Peng et al. [30], Pradel et al. [33], Allamanis et al. [3], doing so may turn the original configuration into a new one that degrades performance significantly. The slowdown can be as high as more than 100 times Takikawa et al. [41], due to intricate type interactions. This is the case even when the type annotations for all the parameters in a single project are inferred. For example, in the spectral norm benchmark, the runtime for the fully typed configuration is about 2 times that of a configuration that has one fewer function typed Campora et al. [8]. The reason is that even all parameters in a project are typed, the libraries and third-party code used by the project may not be typed.

In general, after migrating from configuration K_s to K_e, manually or with the aid of type migration tools, the developer may face a few performance related questions. In particular, if the performance at K_e is not satisfactory, then the user will have to explore the performance of the neighbors of K_e to find a configuration that can restore the performance at K_s or whose performance is the best among all neighbors.

To illustrate, consider the performance lattice for the Pascal benchmark in Figure 1. The Pascal program has 19 parameters and thus 2^{19} configurations, and we present a part of the lattice in the figure. Assume the user is currently at configuration ① and a migration tool infers types for the four parameters, which corresponds to configuration Ⓖ. However, noting that the performance at Ⓖ is about 3 times slower than that at ①, the user will explore the performance of neighbors and find one with good performance.

The problem is that there is no obvious strategy Greenman et al. [15], Takikawa et al. [41] that the user could employ to quickly find desired configurations. For example, a strategy like breadth-first-search will not find Ⓕ, the configuration that both has good performance and has largest number of parameters being typed, without trying Ⓒ, Ⓓ, and Ⓔ. Similarly, a strategy like depth-first-search will not find any configuration that restores performance until it goes back to the original configuration ①. This problem will become worse in practice due to three reasons. First, type migration tools may suggest adding types to many more parameters, which quickly enlarges the search space. Second, as the program becomes bigger, it takes more time to measure the performance of each configuration. Also, it takes more time to move from one configuration to another as more type changes are involved. Third, since each program has its own structure and type of interactions, as witnessed by very diverse performance lattices in different programs Takikawa et al. [41], Greenman et al. [16], Campora et al. [8], no single searching strategy works well for all programs.

The biggest problem is probably the uncertainty associated with the exploration process. If the user has not found a configuration with good performance following some strategy, should the user stick to the strategy in hoping that the performance will finally improve or change the strategy in fearing that performant configurations are in other neighborhoods.

1.2 A Machine Learning Based Solution

In this paper, we propose and develop `LearnPerf`, a machine learning based solution for this problem. For each program, we train a model from the running time for a very limited number (usually 40) of configurations. We then use this model to predict the execution times of other configurations. To give a sense of how the predicted times of `LearnPerf` look like, we present them in Figure 1 in blue.

Our prediction result is pretty accurate, with the *difference ratio* (defined as |predicted time – measured time|/measured time) often within 4%. On top of the prediction result, we can develop a series of migration support under different scenarios. We list some of them below.

1. `LearnPerf` is able to predict the performance for a given configuration. Assume the developer wants to migrate the current configuration to a new one, this information can inform how performance looks like at the new configuration.
2. `LearnPerf` is able to classify the performance of adjacent configurations. For a certain number of configurations around the current one, we can classify them according to performance speedup/slowdown scales. Takikawa et al. [41] introduced the notion 2-deliverable, which includes all configurations whose performance degrades by less than 2 times that of the original configuration, and 2-5 usable, which includes all configurations that slows down the original configuration by 2-5 times. With the help of `LearnPerf`, we can highlight configurations that are 2-deliverable, 2-5 usable, etc.
3. For each configuration, `LearnPerf` is able to find the most performant configurations within its neighborhood. If the user is not satisfied with the performance of the current configuration, this capability can suggest an alternating configuration with good performance.

Note that this work studies the performance aspect of type migration only and is not intended to develop a new type inference algorithm or machine learning algorithm to automatically add type annotations. As mentioned earlier, there has been a long line of research of adding type annotations but little work has been done for the performance aspect except for two papers Campora et al. [8], Greenman et al. [15]. Many approaches Feltey et al. [12], Ortin et al. [28], Moy et al. [25], Kuhlenschmidt et al. [20], Vitousek et al. [46] have investigated performance optimization of gradual typing. Our work is orthogonal to these approaches and we discuss the relation with them in Section 6.

To illustrate the usefulness of our migration support, assume the user was at configuration ① and has just migrated to Ⓖ and observed that the performance at Ⓖ is not satisfactory. `LearnPerf` can help in this case. For example, there are four neighbors of Ⓖ, Ⓒ through Ⓕ. `LearnPerf` predicts that their running times are 19.4, 30.2, 19.2, and 7.3 seconds, respectively. Based on the predicted times, `LearnPerf` suggests the user to migrate to Ⓕ, rather than Ⓖ. We can observe that Ⓕ slows down ① by only 15% while Ⓖ slows down by 3 times.

Overall, `LearnPerf` finds Ⓕ that reconciles both performance and static migration. It seems undesirable to migrate to Ⓕ and not Ⓖ because Ⓖ adds a type annotation to one more parameter. In practice, this problem can often be solved by migrating a few parameters in unison in later migrations.

1.3 Workflow and Contributions of This Work

Figure 2 presents the workflow of `LearnPerf`. Starting from any "original program", assume the user added some type annotations, manually or with the help of type migration tools. This will lead to a new, more precise program that has more type annotations than the original program. Note, that the new program may be partially typed or fully typed. As discussed earlier, both partial and fully typed programs may experience significant performance degradation. From the new program, we create 40 random training samples. We then extract relevant feature vectors that characterizes runtime performance as well as the running time for each generated configuration. This information is then input into `LearnPerf` for the purpose of training our model.

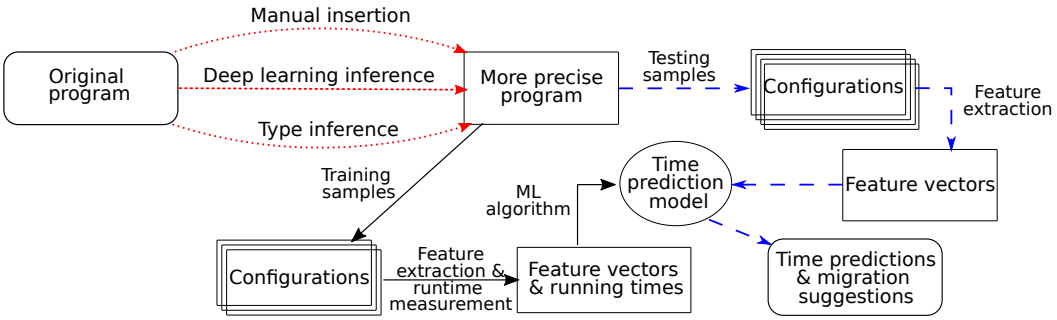

Figure 2 Workflow of `LearnPerf`. Solid arrows denote information flow in training phase and dashed, blue arrows denote that in prediction phase. Dotted, red arrows denote type annotation additions, and they are not a part of this work.

Once we have the trained model, we can support scenarios 1 through 3 by generating appropriate configurations and predicting their running times. Section 5 will sketch the main steps to support scenarios 2 and present our evaluation results.

In the above, the training will not start until the user initiates it. In practice, however, our approach will actively invoke a deep learning model Mir et al. [23], Peng et al. [30], Pradel et al. [33], Allamanis et al. [3] to generate type annotations. As a result, the performance prediction model could be ready before the programmer starts the migration process and needs migration support. We applied this idea to three large datasets for our evaluation (Section 5).

Overall, this paper makes the following contributions:

1. We develop a machine-learning based approach that can help understand the performance landscape of different configurations of a gradually-typed program. On top of that, many migration supports can be developed.
2. We explore different features to represent program run times and find out that overheads of casts inserted by gradual typing are simple yet representative features.
3. We implement our approach and evaluate its performance on twelve benchmarks, including nine benchmarks that are frequently used in gradual typing research and three larger benchmarks that each have more than 1000 LOC. We observe that with only 40 training instances, our predicted times differ from measured times by 4% only for guarded semantics. For transient semantics, the difference between the predicted time and measured time is often within 0.1 seconds.

The rest of the paper is organized as follows. In Section 2, we discuss the background of gradual typing. In Sections 3 and 4, we present our exploration of searching for appropriate machine learning model and representative features for precisely estimating execution times of gradual programs. In Section 5, we present the evaluation results, as well as implementation details and benchmarks used. We discuss related work in Section 6 and conclude in Section 7.

2 Background

This section covers the background of gradual typing, with a focus on cast insertions and their overheads.

In gradual typing, a parameter may be given a static type, a dynamic type (often written as `Dyn` or is omitted) signifying that the type is not known statically, or a mix of static and dynamic types. Static type checking is applied to program parts that use parameters with static types, and dynamic type checking is used for other program parts.

```
def myreduce(f, lst, init):
    result = init
    for i in range(len(lst)):
        result = f(result,lst[i])

    return result

def wider(cw:Int,ci:List(Int))->Int:
    return max(cw, len(ci))

myreduce(wider,[[1], [], [4,5]],0)
```

```
def myreduce(f, lst, init):
    result = init
    for i in range(len(lst)):
        result = (f : Dyn => Dyn -> Dyn -> Dyn)
                 (result, lst[i])
    return result

def wider(cw, ci):
    return max(cw, len(ci)) : Dyn => Int

myreduce(wider: Int -> List(Int) -> Int => Dyn,
         [[1], [], [4,5]] : List(List(Int)) => Dyn,
         0 : Int => Dyn)
```

Figure 3 A partially-typed version of `myreduce` (left) and its cast-inserted program (right).

For example, Figure 3 (left) presents a program snippet written in a hypothetical gradual language in Python type hint syntax Vitousek et al. [45]. The function `myreduce` takes in a binary function, a list, and an initial value and reduces the list to a single value. In this program, static type annotations are given to the parameters, and the return of `wider`. All other parameters have dynamic types. A static type error will be detected if we pass a string value as the first argument to `wider` because the first parameter has a type annotation `Int`. In contrast, no such error will be detected if we pass a string value as the first argument to `myreduce`.

Gradually-typed languages are often obtained by adding static type checking to underlie dynamic languages, such as Typed Clojure for Clojure, Typed Racket for Racket, and Reticulated Python Vitousek et al. [45] for Python. As such, a common implementation strategy of gradual typing is to translate its programs into programs in the underlying language and insert necessary runtime type checks (often called casts) during the translation.

For example, when executed by a gradual typing implementation for Python, the program in Figure 3 (left) is translated to the program in Figure 3 (right), which can be executed on any Python interpreter. Comparing programs in Figure 3 left and right, we observe two important differences. First, the program on the right does not have type annotations. This is because the interpreter for the underlying, dynamic language often does not make use of type annotations so they are erased during translation. Second, the program on the right has extra constructs in the form of `expr : src_type => trg_type`, which are often called casts. Such casts are inserted when the static type checker determines that `expr` has the type `src_type` but is used in a context where a value of `trg_type` is required.

As runtime type checks, these casts incur runtime overheads, and different casts lead to very different overheads. For example, the cast `x : Dyn => Int` can be performed where it appears as we can always verify if `x` is indeed an integer and is thus very lightweight. In contrast, the cast `g : Dyn => Int -> Bool` can not be verified where it appears because, for example, we do not know how `g` will be used and what arguments will be passed to it. As such, a proxy will be created for `g` such that the invocation of `g` will be handled by the proxy, which inserts a cast to check that the argument to `g` is `Int` and another cast to check that the return value of `g` is `Bool`. Such casts are more involved and lead to more significant overheads. Casts over data structures and objects are similarly heavyweight.

It is not hard to envision that adding or removing the type annotation for a single parameter in gradual typing may yield significant performance swings Takikawa et al. [41], Greenman et al. [16]. One might consider this a reason to abandon gradual language designs that enforce type invariants at runtime, but a study by Tunnell Wilson et al. [44] shows that programmers often expected the behavior of programs to emulate those done by gradual typing. This work in this paper enables programmers to enjoy the benefits of gradual typing while staying informed about the performance landscape as they migrate programs toward more static.

Table 1 Deep learning model performance on unseen benchmarks.

Unseen Benchmark	# training	# testing	MAE	MSE	*DR*
Meteor	105945	1024	5.26	28.09	56.84%
Zebrafy	103969	3000	149.37	22350.18	86.96%
Pascal	101785	5184	22.82	653.85	273.42%
Chaos	100969	6000	25.67	784.96	35.57%
Richard	100969	6000	23.73	766.28	36.05%
Sieve	91608	15361	53.88	2933.99	1655.90%
Nbody	90585	16384	4.87	36.79	68.28%
Scimark	81881	25088	4.39	28.27	80.37%
Raytrace	73065	33904	6.49	46.29	110.17%

3 Feature Engineering

The two most important questions in machine learning are what kinds of models to train and what features will be used for representing programs. In this section and next, we present our exploration of searching for a suitable model and simple yet representative features.

3.1 First Attempt: Global Model with Deep Learning

Ideally, we train a *global* model that can be used to predict the runtime of different configurations for all user programs. Such a model needs to be trained only once by us (the model developer) and can be distributed to users (developers who migrate gradual programs) for use.

Motivated by recent successes of deep learning models for predicting types Mir et al. [23], Peng et al. [30], Pradel et al. [33], Allamanis et al. [3], our first attempt is to exploit deep learning to train a global model. For a given set of configurations for training and a set for testing, this process consists of several steps. The main challenge here is that the training instances may have different lengths. To solve this issue, we leverage source code embeddings that convert each configuration into an embedding that has the same length. Specifically, we use UniXcoder Guo et al. [17] to convert each configuration into a 4 * 768 float matrix. These embeddings, together with runtimes of corresponding configurations, are fed into a multi-layer perception network Popescu et al. [32] to train a global model. Based on the trained model, we can predict the runtime for each configuration in the test set.

To test the performance of this idea, we have developed a prototype and conducted experiments in two settings. In the first setting, we collected all configurations from nine benchmarks (listed in Table 1), with a total of 106,969 configurations (Section 5.1 will give more details about our evaluation benchmarks). We randomly choose 80% of them for training, 10% for cross-validation, and 10% for testing. In the second setting, we chose one benchmark for testing and used configurations from all other benchmarks for training. The main difference between these two settings is that in the first setting some testing configurations and training configurations may come from the same benchmark.

To measure the performance of this exploration and later ones in this paper, we use two of the most popular metrics for a regression problem, mean absolute error (*MAE*) and mean square error (*MSE*). In addition, to capture the accuracy or error ratio more intuitively, we used another metric called difference ratio, shortened to *DR*. The definitions of these three metrics are given below, where t_i and $\hat{t}_i$ denote the measured and predicted running times of

the configuration i, respectively, and D denotes the testing set of instances. For example, if the measured and predicted times for a configuration is 7.9s and 8.1s, respectively, then the difference ratio for this configuration is 2.53%. We will use these notations throughout the paper.

$$MAE = \frac{\sum_{i=1}^{D} |t_i - \hat{t_i}|}{|D|} \qquad MSE = \frac{\sum_{i=1}^{D} (t_i - \hat{t_i})^2}{|D|} \qquad DR = \frac{\sum\limits_{i \in D} \frac{|t_i - \hat{t_i}|}{t_i}}{|D|}$$

The DR for the first setting is 147.58%. The details of the results for the second setting is given in Table 1. The results show that the global model trained with deep learning performs poorly. There are a few possible reasons. First, as discussed in Section 2, a gradual program is often translated to a base program in the untyped, underlying language with casts inserted. As such, the running time of a configuration roughly includes the time to execute the base program and the overhead due to casts. To be able to precisely predict the running time, we need to be able to do that for both parts. However, predicting the running time of a general program is still an open problem Matsunaga and Fortes [21]. Second, the overhead due to casts can vary significantly across different programs as it depends on program structures, such as whether casts are in loops, whether multiple casts are applied to single values, etc. Third, as discussed in Section 2, two configurations that differ by whether a single parameter is typed or not may have very different runtimes. This exhibits similar phenomena as in molecular property prediction where minor changes in molecular structures lead to significant changes of properties Stumpfe and Bajorath [40]. Earlier work Xia et al. [50] has demonstrated that deep models often do not perform well for such tasks.

For this reason, we decide to train an individual, project-specific model for each project in this work. Have decided which model to train, we next explore different feature representations to find representative features.

3.2 Second Attempt: Individual Models with Bit Strings

The problem of predicting gradual typing performance bears some similarity to performance prediction for highly-configurable software systems Kolesnikov et al. [18]. A highly-configurable program usually contains a large number of configuration options (for example, Linux has about 13,000 such options) for customizing the functional and non-functional features of the program. For instance, Linux can be configured to run on a diverse set of devices, ranging from embedding devices to servers. Each configuration option may be set or unset, corresponding to enabling or disabling associated features, which often leads to the inclusion or exclusion of certain pieces of code into the generated program after customization. As such, different configurations of the same configurable program will lead to different performances.

Understanding the performance landscape of configurable software systems is an important research problem, particularly as generating all possible programs and measuring their performance is infeasible due to the exponential complexity (the number of different configurations that can be generated is exponential in the number of configuration options). A prevalent solution to this problem is building a *performance-influence* model for each configurable software system. This can be achieved by generating a few samples, measuring the performance of these samples, and building a model from them. With the *performance-influence* model, predicting the performance of a certain configuration is instantaneous, without having to generate the configuration and measure the performance.

Table 2 Python benchmark Performance (Bit strings).

Benchmark	# training	# testing	MAE	MSE	*DR*
Monte Carlo	40	344	0.53 ± 0.00	0.45 ± 0.00	35.30%
Meteor	40	984	0.29 ± 0.02	0.19 ± 0.068	2.47%
CPU	40	2857	2.57 ± 0.06	3.487 ± 0.08	8.15%
Zebrafy	40	3960	12.25 ± 1.24	15.98 ± 0.78	91.64%
Pascal	40	5144	5.15 ± 0.18	6.44 ± 0.31	27.57%
Chaos	40	5960	2.38 ± 0.04	3.00 ± 0.06	4.78%
Richard	40	5960	9.58 ± 1.12	13.65 ± 1.68	42.88%
BenchFirst	40	5960	43.85 ± 3.88	58.58 ± 4.23	25.88%
Sieve	40	15321	0.12 ± 0.00	0.16 ± 0.00	1.56%
Nbody	40	16344	3.45 ± 0.17	4.46 ± 0.19	30.10%
Scimark	40	25048	2.41 ± 0.03	3.06 ± 0.05	17.02%
Raytrace	40	33864	5.80 ± 1.23	7.35 ± 1.60	38.90%
Monte Carlo	192	192	0.39 ± 0.00	0.47 ± 0.00	31.33%
Meteor	512	512	0.13 ± 0.00	0.08 ± 0.00	0.70%
CPU Benchmark	1427	1428	2.23 ± 0.03	2.84 ± 0.02	6.55%
Zebrafy	2000	2000	10.72 ± 0.95	14.58 ± 0.89	87.24%
Pascal	2592	2592	3.81 ± 0.00	5.09 ± 0.00	20.45%
Chaos	3000	3000	1.72 ± 0.01	2.13 ± 0.00	3.41%
Richard	3000	3000	8.86 ± 0.01	13.05 ± 0.02	36.78%
BenchFirst	3000	3000	28.85 ± 3.05	36.58 ± 3.90	11.88%
Sieve	7680	7680	0.10 ± 0.00	0.13 ± 0.00	1.32%
Nbody	8192	8192	2.70 ± 0.00	3.58 ± 0.00	23.64%
Scimark	12544	12544	1.75 ± 0.00	2.42 ± 0.00	12.36%
Raytrace	16952	16952	2.81 ± 0.00	3.19 ± 0.00	18.84%

In gradual typing, each parameter can be typed or untyped, corresponding to enabling or disabling a configuration option. Due to this similarity, we started our exploration by using bit-string as features for machine learning. Specifically, we treat each parameter as a binary feature and use 1 to denote that the parameter is typed and 0 to denote it is untyped. Feature values for all parameters are concatenated together to form a bit-string, which forms the feature vector in this exploration.

We developed a prototype implementing this idea and tested its performance on 12 Python benchmarks (we will show details about them in Section 5.1). We present the result in Table 2. In the upper part of Table 2, we present the results with bit-strings as features when each individual model is trained with 40 configurations. We can observe that the average difference ratio (*DR*) is quite high for several benchmarks. For example, *DR* is around 92% for Zebrafy and 43% for Richard. We may think of increasing the number of training instances to boost the performance. Surprisingly, the performance does not increase significantly as we remarkably increase the number of training instances, as can be seen from the bottom part of Table 2. For example, as we increased the training instances from 40 to 2000 (that is we used 50% of instances for training) for Zebrafy, the average *DR* is still around 87%. Similarly, the average *DR* is about 37% for Nbody as we use 50% of all instances for training.

Another issue is that as we are training an individual model for each project, using too many training instances needs a very long preparation time. To solve this issue, we choose to generate a limited amount of configurations but extract highly effective features.

```
def myreduce(f:Function([Int,List(Int)],Int),
             lst:List(List(Int)), init:Int):
  result = init
  for i in range(len(lst)):
    result = f(result,lst[i])
  return result

def wider(cw:Int, ci:List(Int)) -> Int:
  return max(cw, len(ci))

myreduce(wider,[[1], [], [4,5]],0)
```

```
def myreduce(f, lst, init):
  result = init
  for i in range(len(lst)):
    result = f(result : Dyn => Int,
               lst[i]) : Int => Dyn
  return result

def wider(cw, ci):
  return max(cw, len(ci)) : Dyn => Int

myreduce(wider, [[1], [], [4,5]], 0)
```

Figure 4 The fully-typed version of `myreduce` (left) and its cast-inserted translation (right).

4 Third and Successful Attempt: Gauging Cast Overheads

The main reason that bit strings do not work well is that bits only represent whether parameters are typed or not while the types of parameters interact in an intricate way. This makes bit strings a poor candidate for capturing inserted casts, which are the main causes for performance overheads. For example, if we compare the programs in Figures 3 and 4, we can observe that while the program in Figure 4 (left) has strictly more type annotations than that in Figure 3, no such relation appears for the casts in the translated programs. In particular, these programs share only one common cast (the cast for the return value in `wider`), and all other casts are different. The running times of these two versions of `myreduce` are very different: the running time of the partially-typed version (Figure 3) is about 16 times that of the fully-typed version (Figure 4). In practice, removing or adding the type for a single parameter may lead to a completely different cast being inserted.

Thus, instead of using bit strings, we will next explore the inserted casts of the translated programs by gauging cast overheads. Our main idea is to give symbolic overheads to casts and let machine learning algorithm figure out the real overhead of each cast. To give a more formal account of our approach, we present the type syntax used for the rest of this section below.

$$\begin{array}{lllll} \text{Base types} & U & ::= & \texttt{Bool} \mid \texttt{Int} \mid \texttt{Unit} \\ \text{Gradual types} & G & ::= & U \mid G \to G \mid \texttt{Dyn} \mid [G] \end{array}$$

Our type definition includes base types, ranged over by U, and gradual types, ranged over by G. Our base types include `Int`, `Bool`, and `Unit`, but they can be extended easily. In gradual types, we consider two type constructors: function types and list types. Again, they can be extended easily.

In the rest of this section, we first discuss how to gauge the overhead for individual casts (Sections 4.1 and 4.2) and then the overhead for a whole program (Section 4.3). Finally, we assess the effectiveness of cast overheads (Section 4.4).

4.1 Overheads for Individual Casts

Casts involving base types. Our first observation of gauging cast overheads is that casts have very different runtime overheads, as we discussed in Section 2. We first deal with casts that involve base types. For a cast of the form $U \Rightarrow \texttt{Dyn}$, it can be checked where it appears. We assign the symbolic overhead U^i to it. Similarly, for the cast $\texttt{Dyn} \Rightarrow U$, we assign the symbolic overhead U^p.

Casts involving function types. Next, we investigate overheads of casts that involve function types. In general, as discussed in Section 2, a function cast can not be verified where it appears. Instead, for a cast of the form $f : G_1 \to G_2 \Rightarrow G_3 \to G_4$, a proxy will be created to

wrap f. In place where f is called, the call is handled by the wrapper, which first casts the argument from G_3 to G_1, calls f with the cast argument, and casts the return value of f from G_2 to G_4. As such, a function cast induces two kinds of overheads: (1) the overhead that creates the proxy and (2) the overhead that casts the arguments and returned values. We refer to these two kinds of overheads as *creation overhead* and *invocation overhead*, respectively. The creation overhead should be similar across different proxy wrappers because type differences in casts do not cause the creation behavior to change much. As such, we assign F^c to represent a proxy creation overhead.

One challenge with invocation overheads is that they are incurred when the cast functions are invoked, not where the function casts appear. However, it is unclear when cast functions are invoked by looking at the translated program (neither with some standard static analysis) because cast functions may be assigned to other variables, stored in data structures, and passed over to other functions, and call sites can be very distant from where proxies are created. Our solution to this problem is to gauge the invocation overhead for each cast and directly add it to its creation overhead. This is very simple to implement: no complex alias analysis is needed.

Interestingly, this approach works well for predicting runtimes of configurations. Intuitively, the function cast created at the same program location will have the same invocation sites across different configurations since two configurations only differ by type annotations. Thus, if two casts cast the same function and have the same invocation overhead across two configurations, then they induce the same cast overheads. Of course, if the arguments to the cast functions in different configurations are cast differently, then the invocation takes different times to complete. However, such differences should be reflected through overhead differences of casts on the argument. Similarly, if the function cast in the first configuration has larger invocation overhead than that in the second configuration, then the cast function in the first configuration has more runtime overheads at invocation sites. We leave it to the machine learning algorithm that we use to train our model to figure out the relation between symbolic difference and the runtime difference for different configurations.

Another challenge in gauging invocation overheads is that unlike creation overheads that are similar across different function casts, invocation overheads can vary significantly, based on the types involved. For example, the cast $f1 : \texttt{Int} \to \texttt{Int} \Rightarrow \texttt{Dyn} \to \texttt{Dyn}$ should have a much smaller invocation overhead than $f2 : [\texttt{Int}] \to \texttt{Int} \Rightarrow \texttt{Dyn} \to \texttt{Dyn}$ because the cast for the argument for $f1$ is $\texttt{Dyn} \Rightarrow \texttt{Int}$ and that for $f2$ is $\texttt{Dyn} \Rightarrow [\texttt{Int}]$. As we have seen earlier, the cast $\texttt{Dyn} \Rightarrow \texttt{Int}$ is very lightweight while the cast $\texttt{Dyn} \Rightarrow [\texttt{Int}]$ involves the creation of another proxy over the argument (We will discuss casts involve lists later in this subsection), which will be treated as a list. Therefore, a plausible idea to accurately gauge invocation overheads is to assign different symbols for denoting different invocation overheads to different casts, based on their argument types and return types. The problem with this idea is that, however, we need to introduce a lot of different symbols for invocation overheads because within a program we could have many casts involving function types with different arities and different argument and return types. As we wanted to train our model with as few instances as possible, having too many symbols will negatively affect machine learning performance.

Our solution to this challenge is to break invocation overheads down and represent them with symbols we have already introduced. Our main insight is that an invocation overhead is originated from creating further casts at runtime. Thus, an invocation overhead can be approximated as a sum of all the creation overheads of the argument types and the return type. For example, for $f3 : (\texttt{Int} \to \texttt{Bool}) \to \texttt{Dyn} \Rightarrow \texttt{Dyn} \to \texttt{Int}$, the invocation overhead is creating a new function proxy for the argument to $f3$, which we have already introduced a symbol F^c, and

another cast for the $\texttt{Dyn} \Rightarrow \texttt{Int}$, which we used U^p to represent the overhead. Since the created function cast for the argument also introduces invocation overhead, we recursively apply this idea to the argument cast $\texttt{Dyn} \Rightarrow \texttt{Int} \rightarrow \texttt{Bool}$ and calculate its invocation overhead as $U^i + U^p$. Overall, the invocation overhead for the function cast $f3 : (\texttt{Int} \rightarrow \texttt{Bool}) \rightarrow \texttt{Dyn} \Rightarrow \texttt{Dyn} \rightarrow \texttt{Int}$ is $F^c + 2 \cdot U^p + U^i$. We give an algorithm for calculating cast overheads in Figure 5.

Casts involving list types. A cast involving list types, such as $l : \texttt{Dyn} \Rightarrow [\texttt{Bool}]$, also can not be verified where it appears because this cast ensures that future write accesses to l should add elements of type `Bool` only and future read accesses should get elements of `Bool` type. As such, similar to casts on function types, a proxy will be created for l and the proxy will make sure accesses to l have expected types. Therefore, the overhead of a list cast includes the creation overhead and access overhead. For the creation overhead, we use the symbol L^c to denote it.

For gauging access overheads, we face a challenge of locating where lists are accessed in the program, some of what we had for gauging invocation overheads for function casts. We adapt the solution there by gauging access overheads and add them to list creation overheads. For gauging access overheads themselves, the main insight is that list accessing can often be reduced to function calls Siek et al. [37], Vitousek et al. [45]. For example, for a list of type $[\texttt{Bool}]$, the function for ensuring that the element read from the list is `Bool` has the type $\texttt{Int} \rightarrow \texttt{Bool}$, where `Int` is the type of the parameter (list index) and `Bool` is the return type. The function for ensuring that the element added to the list is `Bool` has the type $\texttt{Int} \rightarrow \texttt{Bool} \rightarrow \texttt{Unit}$, where `Int` is the index type, `Bool` is the type of the element to be added to the list, and `Unit` is the return type of the function.

Based on this idea, the read access to the list l with the cast $l : \texttt{Dyn} \Rightarrow [\texttt{Bool}]$ can be reduced to the function cast $\texttt{Int} \rightarrow \texttt{Dyn} \Rightarrow \texttt{Int} \rightarrow \texttt{Bool}$, and the write access can be reduced to the cast $\texttt{Int} \rightarrow \texttt{Dyn} \rightarrow \texttt{Unit} \Rightarrow \texttt{Int} \rightarrow \texttt{Bool} \rightarrow \texttt{Unit}$. Thus, the access cost is approximated to be the cost of these two function casts. In practice, other operations may be performed on a list, such as insertion, extension, popping, and obtaining the length. However, read and write accesses are good representatives of access overheads because they are used frequently while others may not need function casts. Moreover, as we did in gauging invocation overheads, we only need to figure out the symbolic difference of list casts for the same list across different configurations, and let the machine learning algorithm scale that difference to appropriate runtime differences.

4.2 An Algorithm for Gauging Individual Casts' Overheads

We present an algorithm for gauging cast overheads in Figure 5. The algorithm is more general than our description in Section 4. For example, the algorithm deals with function casts that have multiple parameters. The algorithm is defined using the idea of pattern matching, and we assume that the most specific matching rule is used to handle the computation.

The main entry of the algorithm is the function *overHd*, which consists of eight cases. In the first case, the two types being cast are the same. Standard gradual typing implementations simply drop such casts, and so we assign 0 as its overhead. Cases two and three deal with casts between `Dyn` and function types, and we extend `Dyn` into a function type with the same arity as the function on the other side and delegate the computation to case eight of *overHd*. Cases four and five deal with casts between two function types that have different number of parameters. We assume that corresponding parameter types (such as G_1 and G'_1) and return types are consistent Siek and Taha [38]. We extend the type with fewer parameter types by padding it with `Dyn`s. Cases six and seven deal with casts between `Dyn` and list types and are

$$
\begin{aligned}
&\mathit{overHd}(G \Rightarrow G) = 0\\
&\mathit{overHd}(\texttt{Dyn} \Rightarrow G_1 \to \cdots \to G_r) = \mathit{overHd}(\texttt{Dyn} \to \cdots \to \texttt{Dyn} \Rightarrow G_1 \to \cdots \to G_r)\\
&\mathit{overHd}(G_1 \to \cdots \to G_r \Rightarrow \texttt{Dyn}) = \mathit{overHd}(G_1 \to \cdots \to G_r \Rightarrow \texttt{Dyn} \to \cdots \to \texttt{Dyn})\\
&\mathit{overHd}(G_1 \to \cdots \to G_i \to \texttt{Dyn} \Rightarrow G'_1 \to \cdots \to G'_{i+j} \to G'_r)\\
&\qquad = \mathit{overHd}(G_1 \to \cdots \to G_i \to \texttt{Dyn} \to \cdots \to \texttt{Dyn} \Rightarrow G'_1 \to \cdots \to G'_{i+j} \to G'_r)\\
&\mathit{overHd}(G_1 \to \cdots \to G_{i+j} \to G_r \Rightarrow G'_1 \to \cdots \to G'_i \to \texttt{Dyn})\\
&\qquad = \mathit{overHd}(G_1 \to \cdots \to G_{i+j} \to G_r \Rightarrow G'_1 \to \cdots \to G'_i \to \texttt{Dyn} \to \cdots \to \texttt{Dyn})\\
&\mathit{overHd}([G] \Rightarrow \texttt{Dyn}) = \mathit{overHd}([G] \Rightarrow [\texttt{Dyn}])\\
&\mathit{overHd}(\texttt{Dyn} \Rightarrow [G]) = \mathit{overHd}([\texttt{Dyn}] \Rightarrow [G])\\
&\mathit{overHd}(G_1 \Rightarrow G_2) = \mathit{createOH}(G_1 \Rightarrow G_2) + \mathit{callOH}(G_1 \Rightarrow G_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{createOH}(\texttt{Dyn} \Rightarrow U) = U^p\\
&\mathit{createOH}(U \Rightarrow \texttt{Dyn}) = U^i\\
&\mathit{createOH}(G_1 \to \cdots \to G_r \Rightarrow G'_1 \to \cdots \to G'_r) = F^c\\
&\mathit{createOH}([G_1] \Rightarrow [G_2]) = L^c
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{callOH}(\texttt{Dyn} \Rightarrow U) = 0\\
&\mathit{callOH}(U \Rightarrow \texttt{Dyn}) = 0\\
&\mathit{callOH}(G_1 \to \cdots \to G_n \to G_r \Rightarrow G'_1 \to \cdots \to G'_n \to G'_r)\\
&\qquad = \textstyle\sum_1^n \mathit{overHd}(G'_i \Rightarrow G_i) + \mathit{overHd}(G_r \Rightarrow G'_r)\\
&\mathit{callOH}([G_1] \Rightarrow [G_2])\\
&\qquad = \mathit{overHd}(\texttt{Int} \to G_1 \Rightarrow \texttt{Int} \to G_2) + \mathit{overHd}(\texttt{Int} \to G_1 \to \texttt{Unit} \Rightarrow \texttt{Int} \to G_2 \to \texttt{Unit})
\end{aligned}
$$

Figure 5 An overhead gauging algorithm.

similarly delegated to case eight. Case eight deals with all cases not matched by earlier cases. It says that the overhead is an addition of the creation overhead, returned from *createOH*, and the call overhead, returned from *callOH*.

The definition of *createOH* is straightforward: it assigns a corresponding symbolic overhead to each kind of cast. The function *callOH* implements the idea of invocation overheads and access overheads discussed in Section 4. For casts involving base types, the call overhead is 0 because they can not be invoked or no elements may be accessed from them. The call overhead for a function cast is the overhead of casting all parameter types plus that of casting the return type. The call overhead for a list cast is the total overhead of read access and write access.

The following example illustrates the calculation process for gauging the overhead for the cast $\texttt{Dyn} \Rightarrow [\texttt{Bool}]$.

$$
\begin{aligned}
&\mathit{overHd}(\texttt{Dyn} \Rightarrow [\texttt{Bool}])\\
=&\mathit{overHd}([\texttt{Dyn}] \Rightarrow [\texttt{Bool}])\\
=&\mathit{createOH}([\texttt{Dyn}] \Rightarrow [\texttt{Bool}]) + \mathit{callOH}([\texttt{Dyn}] \Rightarrow [\texttt{Bool}])\\
=&L^c + \mathit{callOH}([\texttt{Dyn}] \Rightarrow [\texttt{Bool}])\\
=&L^c + \mathit{overHd}(\texttt{Int} \to \texttt{Dyn} \Rightarrow \texttt{Int} \to \texttt{Bool}) + \mathit{overHd}(\texttt{Int} \to \texttt{Dyn} \to \texttt{Unit} \Rightarrow \texttt{Int} \to \texttt{Bool} \to \texttt{Unit})\\
=&L^c + F^c + \mathit{overHd}(\texttt{Int} \Rightarrow \texttt{Int}) + \mathit{overHd}(\texttt{Dyn} \Rightarrow \texttt{Bool}) + \mathit{overHd}(\texttt{Int} \to \texttt{Dyn} \to \texttt{Unit} \Rightarrow \texttt{Int} \to \texttt{Bool} \to \texttt{Unit})\\
=&L^c + F^c + 0 + U^p + F^c + \mathit{overHd}(\texttt{Int} \Rightarrow \texttt{Int}) + \mathit{overHd}(\texttt{Bool} \Rightarrow \texttt{Dyn}) + \mathit{overHd}(\texttt{Unit} \Rightarrow \texttt{Unit})\\
=&L^c + F^c + 0 + U^p + F^c + U^i\\
=&L^c + 2 \cdot F^c + U^p + U^i
\end{aligned}
$$

Due to the limited space, the algorithm in Figure 5 deals with base types, function types, and list types only. Our implementation supports many more types, including dictionary types, tuples, objects, records, and several others, with the same idea.

4.3 Representing Overheads for a Program

Without the loss of generality, we assume that a program consists of a few functions and top-level statements. When the program is translated, casts are inserted into function definitions and top-level statements. To extract the feature vector for a program, we repeat the following for each function. For each cast inserted in the function, we use Figure 5 to calculate the overhead. We then sum the overheads for all casts together. If a cast appears in a loop, then we automatically instrument the loop, obtain the number of times the loop is executed, and multiply the cast overhead by that number. For example, if a function has two casts that are outside of loops and have the overheads $F^c + U^p$ and $L^c + F^c + U^p + U^i$, then the total overhead for that function is $L^c + 2 \cdot F^c + 2 \cdot U^p + U^i$. The feature vector for that function is the coefficients of all overhead symbols, represented as $1, 2, 2, 1$ in this case. The machine learning algorithm will turn these coefficients into runtime predictions.

Similarly, for the casts inserted in top-level statements, we calculate the overhead of each cast and sum them together.

Finally, we concatenate representations for all functions and top-level statements, forming a list of coefficients. This list will be the feature representation of the whole program.

4.4 Assessing Feature Effectiveness

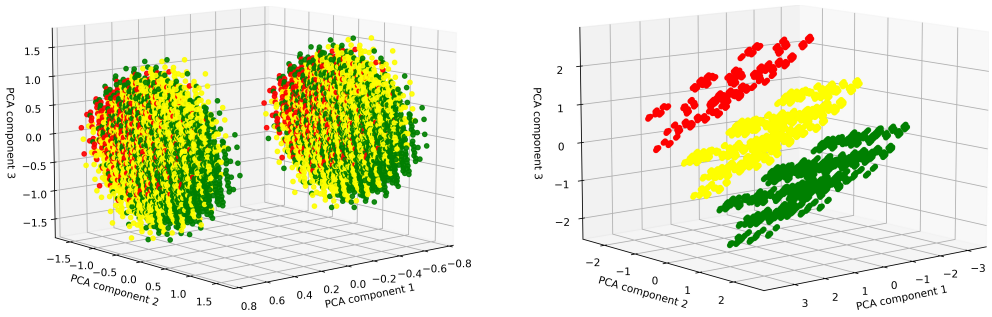

Figure 6 3D PCA analysis for Nbody using bit strings (left) and cast overheads (right). Both figures are generated with elevation of 10.0 and use azimuth angle 50.

Our approach `LearnPerf` is developed using cast overheads as features. For all the benchmarks we used to evaluate the performance, the *DR* is always less than 4% except for one benchmark whose *DR* is 5.3% (We will present the results in more detail in Section 5). In general, this means that our predicted time is in average within 4% of difference compared to the real measured time. We view this as a significant improvement over the performance of bit-string based solution, where *DR* is often higher than 30% and can be as high as 90%.

We have performed a PCA analysis Abdi and Williams [1] to understand the effectiveness of both bit strings and cast overheads. Figure 6 presents the analysis results for Nbody. In the figure, axes represent values of PCA components and colored circles represent configurations. In particular, configurations with similar running times get the same color. The running times of Nbody are roughly in three groups: those less than 7.5s (seconds), between 12.5s and 15s, and more than 20s (see Figure 7 for more details). From Figure 6, we can observe that bit strings fail to separate configurations while cast overheads successfully separate configurations according to their runtimes. Intuitively, clear separations of configurations according to their runtimes mean fewer prediction errors. This shows the usefulness of using cast overheads as features.

Table 3 Python benchmarks used for performance evaluation. The last column gives the number of configurations generated for the corresponding benchmark.

Benchmark	LOC	# of functions	# of pars	# of typed pars	# of configurations
Monte Carlo	90	4	9	9	385
Meteor	238	8	26	14	1024
CPU	2824	32	39	23	2897
Zebrafy	1578	28	72	38	4000
Pascal	70	7	19	15	5184
chaos	271	22	42	29	6000
Richard	455	21	94	67	6000
BenchFirst	1017	27	76	54	6000
Sieve	56	9	22	21	15361
Nbody	195	4	21	18	16384
Scimark	65	5	22	17	25088
Raytrace	254	37	67	38	33904

5 Performance Evaluation

We have implemented `LearnPerf` in Python. The main components are type addition, feature extraction, model training. Some of evaluated benchmarks are adopted from earlier work in gradual typing Campora et al. [8], Vitousek et al. [45], which already have type information. For other benchmarks, we use HiTyper (Peng et al. [30]), a state-of-the-art deep learning approach, to infer types that may be added. One issue with HiTyper is that some inferred types are erroneous, as noted by Yee and Guha Yee and Guha [51]. We remove a type annotation whenever adding it causes static type conflicts. We generate a new, more precisely typed program after merging the type annotations from HiTyper into the original program. From the new program, we generate a desired number of configurations for each benchmark (Table 5).

We implemented feature extractions on top of Reticulated Python (Vitousek et al. [45, 47, 46]). To test the generality of our approach, we have implemented feature extractions for both the guarded semantics Vitousek et al. [45] and transient semantics (Vitousek et al. [47]). Since these two semantics lead to different translated programs, we have different feature extraction codes. However, both implementations are based on the idea of gauging cast overheads, discussed in Section 4. Our feature extraction, which totals about 1,850 lines of code, supports the most commonly used Python types, including lists, functions, dictionary types, tuples, iterables, objects, classes, and many others.

The model training component is implemented on top of the scikit-learn Pedregosa et al. [29] package, a frequently used machine learning Library in Python. We use scikit-learn's various models, its training-testing data split package, and its metrics package. This component includes less than 200 LOC.

5.1 Benchmarks

To evaluate the performance of `LearnPerf`, we adopted nine benchmarks that were commonly used in gradual typing research in Python (Vitousek et al. [47, 46], Campora et al. [8]. These programs are relatively small, often below 500 LOC. In addition, we adopted three large benchmarks, including Zebrafy (a Python program for creating PDF files) and CPU

Benchmark and BenchFirst (two performance bench-marking programs). For each benchmark, we present the name, lines of code, number of functions, number of parameters, number of parameters that are typed originally or with the help of HiTyper, and total number of configurations we generated for evaluating our performance in Table 3.

The number of configurations generated for each benchmark is mainly determined by two factors: the number of parameters in the benchmark and the time required to run each configuration. For example, each configuration in Zebrafy, CPU Benchmark, and BenchFirst takes more than 100 seconds to finish. As a result, we generate about only 4000 configurations for such benchmarks.

The configurations for each benchmark for evaluating performance are generated follow the insights from Greenman et al. [16] to ensure that they are representative. We can imagine that all configurations from a benchmark be organized into a lattice based on the parameters that are typed. The lattice includes 2^n configurations if n parameters are typed. All configurations in the same row of the lattice assign types to the same number of parameters. For example, the bottom-most row assigns types to zero parameters, and the row above assign types to only one parameters, and the row above that assign types to two parameters, and so on.

In our experiments, we generated configurations such that every row of the lattice is covered. Moreover, we try to maintain same proportion of generated configurations over all configurations in a row across all rows. However, for middle rows, the percentage is smaller because there are too many configurations in them. For example, the middle row has $C_n^{\frac{n}{2}}$ configurations. Once these configurations are generated, we randomly split them so that 40 are used for training and the remaining are used for testing. Note, we repeated 5 times for the training/testing process.

The running times in this paper are measured on a machine equipped with Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz, 8 Core(s), and 16GB RAM. Each measured time is an average of 10 runs.

Figure 7 gives an idea of how execution times look as a certain number of parameters are typed. The figure shows that while the running times of some benchmarks are clustered, others are scattered. We believe that these benchmarks serve the evaluation purpose well.

Our evaluation focus on Scenarios 1 and 3 only. The result for Scenario 2 is similar to that for Scenario 3, and we omit it in the paper.

5.2 Supporting Scenario 1

To simulate the real development scenario, we randomly selected 40 instances from all generated configurations as training instances, and we use linear regression to train a time prediction model. Compared to standard machine learning applications, our approach uses significantly fewer data instances for training.

To ensure that the model correctly learns patterns from the data and doesn't pick up too much noise, we used k-fold cross-validation technique. As is standard in machine learning practice, our results are averaged over all k trials to get the overall performance of the model. We set k to 5 in our evaluation. Experiments were run on the same machine we used for generating benchmark's configurations.

Table 4 describes the performance of `LearnPerf` on all evaluated benchmarks. Columns three through five of the table show that even when the model is trained with only 40 instances, our prediction result is very accurate, with *DR* (defined in Section 3.2) less than 3% for nine benchmarks, between 3% and 4% for two benchmarks, and is 5.26% for one benchmark. Intuitively, this means that our predicted times are very close to measured times.

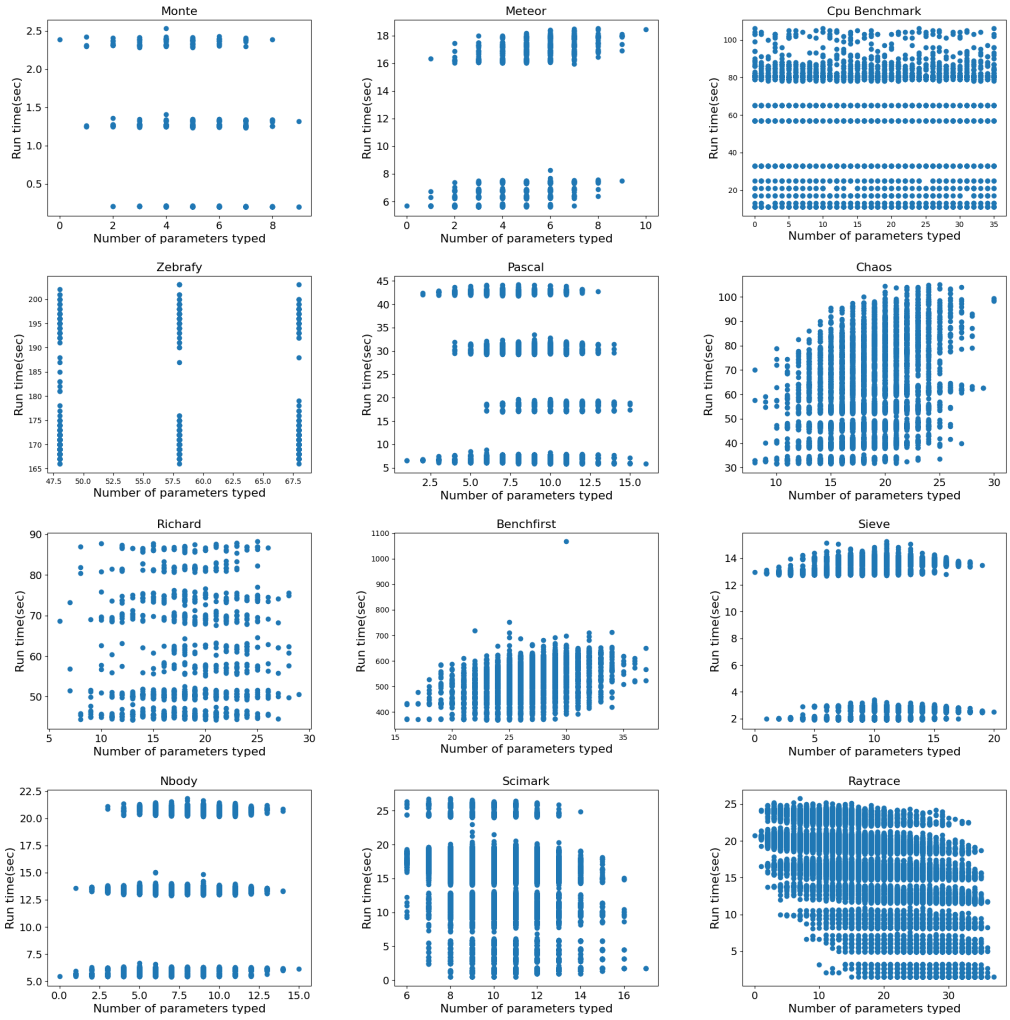

Figure 7 Benchmark's configurations description: Run Time vs Number of parameters typed for each benchmark.

Columns six and seven of Table 4 present the ratios of configurations whose *DR* are less and greater than 10%, respectively. The result shows that there are fewer configurations that have large difference ratios.

Figure 8 presents a closer investigation of the evaluation result. Specifically, we divide each benchmark into five groups in terms of their measured running time of different configurations of the benchmark. Next, we predict the performance (running time) and measure the *DR* of all configurations within each group. For every group, green represents a *DR* of less than 5%, cyan represents 5 to 10%, blue represents 10 to 15%, violet represents 15 to 20%, and red represents more than 20% of *DR*. The figure reveals that, in general, the configurations that have smaller running times tend to experience higher *DR*s. There are two potential reasons behind this. First, a small variance in predicted time for such configurations can lead to a higher *DR*. Second, even averaged over ten runs, each measured time includes a small randomness due to computer execution dynamics, and the randomness in such configurations has a more conspicuous impact.

Table 4 The performance of `LearnPerf` on evaluated benchmarks. The model for each benchmark is trained with forty randomly selected configurations. The second column gives the number of testing instances (configurations). Columns three through five gives the average performance of all testing instances. Columns six and seven give the ratios of instances whose *DR* are less than and greater than 10%, respectively.

Benchmark	# testing	MAE	MSE	*DR*	<10%	>10%
Monte Carlo	344	0.05 ± 0.01	0.07 ± 0.01	3.597%	93.77%	6.23%
Meteor	984	0.32 ± 0.01	0.39 ± 0.02	2.70 %	97.35%	2.65%
CPU	2857	1.56 ± 0.01	2.07 ± 0.08	1.91%	99.80%	0.2%
Zebrafy	3960	2.70 ± 0.00	3.63 ± 0.00	1.57%	92.88%	7.12%
Pascal	5144	0.37 ± 0.03	0.460 ± 0.04	2.10%	95.55%	4.45
Chaos	5960	2.37 ± 0.07	2.92 ± 0.09	3.56%	97.09%	2.91%
Richard	5960	0.55 ± 0.00	0.71 ± 0.00	0.91%	100%	0.0%
BenchFirst	5960	17.12 ± 0.03	25.06 ± 0.7	5.26%	86.04%	13.96%
Sieve	15321	0.17 ± 0.01	0.23 ± 0.01	2.18%	89.06%	10.94%
Nbody	16344	0.21 ± 0.01	0.25 ± 0.01	1.84%	99.86%	0.14%
Scimark	25048	0.14 ± 0.04	0.193 ± 0.05	0.97%	98.92%	1.08%
Raytrace	33864	0.26 ± 0.06	0.330 ± 0.09	1.73%	94.46%	5.54%

5.3 Supporting Scenario 3

Scenario 3 aims to find the neighbor with best performance for any given configuration. This is particularly helpful when the current configuration has poor performance and the user wants to find a neighbor with good performance.We can imagine that there are two lattices with the current configuration.One grows up, adding more type annotations to current configuration, and one grows down, removing type annotations from the current configuration. We then use the idea of these two method to choose neighbors. However, here we consider configurations that add/remove up to seven parameters. To evaluate how well `LearnPerf` can support this scenario, we randomly choose a certain number of configurations, and find the most performant neighbor of it using our model.

We present the detailed result for this scenario in Table 5, which includes the number of configurations considered as the current configuration (the second column) and three metrics to measure the performance of `LearnPerf`. To simplify our discussion below, we refer to a configuration and all its neighbors as a *region*. Each region includes at least 100 neighbors or includes all neighbors that add types to up to seven parameters. The first metric (column three in the table) is the accuracy. For any given configuration, if the most performant neighbor identified by `LearnPerf` is among the three neighbors with least execution times, then we classify this as a correct identification. We consider top three neighbors because it is common for many neighbors to have very small difference in execution times. The accuracy is calculated by dividing the number of correct identifications over all regions considered for that benchmark. For example, for Scimark, we considered 500 regions, and for 408 regions `LearnPerf` made correct identifications. As a result, the accuracy is 81.6%.

The second metric (column four in the table) is the average differences between the execution times of the real and the identified most performant neighbors. For example, if the real most performant neighbor for a region has an execution time of 4.73s and the identified neighbor has an execution time of 4.75s, then the time difference is 0.02s. This column records the average of differences of all regions for that benchmark. The third metric (column five in the table) calculates the time difference in percentage.

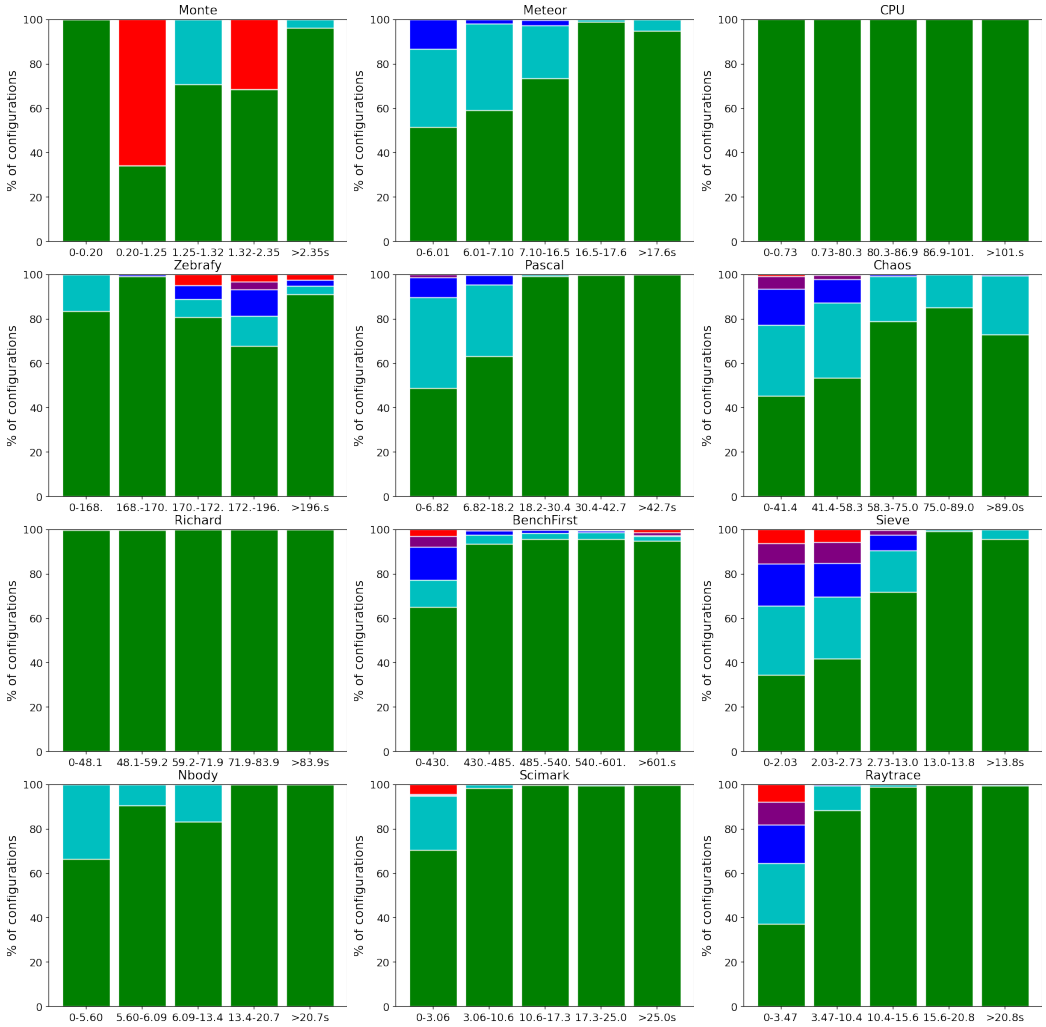

Figure 8 `LearnPerf`detailed benchmark's performance based on different measured run time groups.

Again, the table shows that our approach is very accurate in identifying the most performant neighbors, with the difference ratio always below 1% except for Pascal that has a 2.4% *DR*.

5.4 Training and Prediction Times

Table 6 presents times needed for generating and measuring 40 configurations for training the model, the time for training the model once these 40 configurations are ready, and the average feature extraction time for each program. We do not present the prediction time because that is less than 1ms for each configuration. From the table, we can see that the most time in our approach is spent on measuring the running times for training the model.

For some benchmarks, measuring the times is relatively fast, such as for Monte Carlo, Meteor, Sieve, Nbody, Scimark, and Raytrace. However, it takes significantly longer to measure the times for some benchmarks, including CPU, Zebrafy, Chaos, Richard, and BenchFirst. The reason is that each configuration from these benchmarks takes a long time to complete. Usually, this large amount of measuring time will lead to a long response time. Also, it looks like this long waiting time is hard to avoid.

Table 5 `LearnPerf`'s performance on finding the most performant neighbor to migrate for each benchmark.

Benchmark	# of regions	Accuracy	Average difference(s)	difference ratio
Monte Carlo	42	100%	0.0	0%
Meteor	500	77.0%	0.032	0.338%
CPU	38	94.74%	0.004	0.998%
Zebrafy	89	98.88%	0.007	0.087%
Pascal	500	44.80%	0.171	2.388%
chaos	297	83.16%	0.059	0.88%
Richard	98	100%	0	0%
BenchFirst	113	93.81%	0.004	0.058%
Sieve	500	63.6%	0.020	0.685%
Nbody	500	34.60%	0.116	1.246%
Scimark	500	81.6%	0.021	0.344%
Raytrace	385	96.88%	0.007	0.034%

Table 6 Training and Prediction time of Each benchmark.

Benchmark	Measuring 40 configurations (s)	Training(s)	Feature extraction (ms)
Monte Carlo	53.27	1.00	10.98
Meteor	490.06	0.99	23.38
CPU	2997.87	3.3	1001.96
Zebrafy	7394.38	4.75	1012.30
Pascal	580.05	1.01	40.89
Chaos	2654.87	1.03	29.15
Richard	2462.77	1.99	1013.33
BenchFirst	21816.94	3.89	1112.32
Sieve	373.33	1.02	19.67
Nbody	488.94	0.99	25.86
Scimark	555.68	0.99	27.73
Raytrace	623.55	2.98	26.17

Fortunately, with the help of type migration tools, we can significantly shorten the response time. The idea is that we start to measure the runtimes way before the user needs the migration support. We tested this idea by automating the process of generating type information for parameters with HiTyper, merging the generated type information into the original program, randomly generating configurations for training, running all generated configurations to measure their runtime duration, extracting features for these configurations, and training a time prediction model based on the collected times and extracted features. We tested this idea on three large benchmarks, including CPU, Zebrafy, and BenchFirst.

Once the model has been trained, predicting the running time is very fast. Since feature generation is also very efficient, we can quickly provide migration support with the model. For example, for any given configuration, `LearnPerf` is able to find the most performant neighbor within a few seconds.

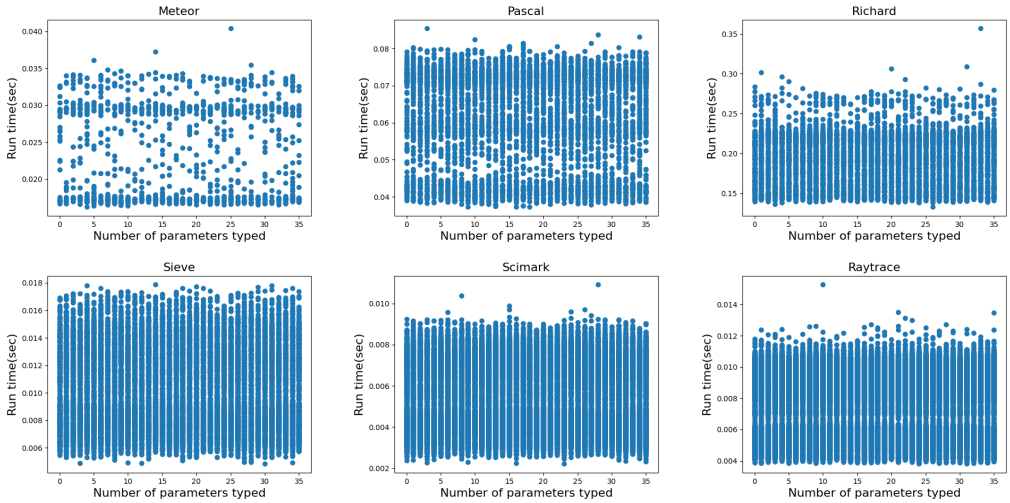

Figure 9 The relation between run times and the number of parameters that have type annotations for six benchmarks for transient semantics.

5.5 Different Machine Learning Methods

We used linear regression to train our model. During the development of `LearnPerf`, we also explored other machine learning algorithms, including random forest regression, decision tree regression, and AdaBoost regression. We decided to use linear regression for the following reasons. First, linear regression usually does not need too many training instances. In our case, 40 training instances yield good performance. Second, training and prediction with linear regression is very fast than other models. Third, linear regression yields good performance across all benchmarks. For example, while random forest achieves 1.38% and 1.19% *DR*s for Monte and Sieve, respectively, the *DR*s for Raytrace and Scimark are above 13%.

We also tried some other famous Machine learning models, such as support vector machine regression and MLP regression, but they either need more training instance or take more times for training and prediction. Also, they do not outperform linear regression for our problem.

5.6 Evaluation of Transient Semantics

In addition to evaluating the performance of `LearnPerf` on the guarded semantics, we have also evaluated it on the transient semantics (Vitousek et al. [47]) using the same benchmarks. In our feature extraction code for transient semantics, we set all the return values of $callOH(\cdot)$ in Figure 5 to 0 because transient casts do not introduce proxies.

Figure 9 presents the runtime distributions for six benchmarks under transient semantics. We omit the other six because their distributions are similar. Comparing this figure to Figure 7 we observe that the runtimes in transient semantics are several magnitudes smaller than in guarded semantics. Also, the runtimes have smaller variations across different configurations.

Table 7 presents the performance of `LearnPerf` for the transient semantics. We can observe that the *DR* is much higher than that for guarded semantics. Meanwhile, we observe that the *MAE* and *MSE* are close to 0. This indicates that a possible reason that *DR* is relatively high because the runtime of each configuration is very smaller, usually below 0.1 seconds. A small randomness in measured time can lead to a high *DR* in this case.

Table 7 Overall performance of `LearnPerf` for the transient semantics.

Benchmark	# training	# testing	MAE	MSE	*DR*
Monte Carlo	40	344	0.002 ± 0.0	0.004 ± 0.0	31.43%
Meteor	40	984	0.005 ± 0.01	0.006 ± 0.0	25.09 %
CPU	40	2857	0.01 ± 0.0	0.02 ± 0.08	18.91%
Zebrafy	40	3960	0.001 ± 0.0	0.003 ± 0.0	27.77%
Pascal	40	5144	0.01 ± 0.0	0.013 ± 0.0	18.549%
Chaos	40	5960	0.02 ± 0.0	0.06 ± 0.0	34.38%
Richard	40	5960	0.027 ± 0.0	0.033 ± 0.0	24.93%
BenchFirst	40	5960	0.033 ± 0.0	0.087 ± 0.0	25.88%
Sieve	40	15321	0.002 ± 0.0	0.001 ± 0.0	29.07%
Nbody	40	16344	0.003 ± 0.0	0.005 ± 0.01	21.78%
Scimark	40	25048	0.001 ± 0.0	0.001 ± 0.0	27.129%
Raytrace	40	33864	0.001 ± 0.0	0.002 ± 0.0	25.497%

Overall, our approach works pretty well for transient semantics also. The main insight is that the algorithm in Figure 5 derives coefficients of cast overheads rather than the real runtime overhead of casts. The overhead of a transient cast (checking type tags) can also be estimated using coefficients.

5.7 Threats to Validity

It may be possible that the results observed in our evaluation do not transfer to other Python programs. We have done the following to reduce this possibility. (1) We chose the benchmarks that are commonly used in the literature for evaluating gradual typing performance as well as three large Python programs (details in Section 5.1). (2) The evaluated programs cover most commonly used language features in Python, including control structures such as conditionals and loops, functions, classes with inheritance, tuples, dictionaries, nested lists, etc. (3) The amount of typed parameters can have a big impact on the results. As shown in Table 5, the percentages that parameters have types are quite diverse, ranging from about 50% (Meteor and Zebrafy) to about 100% (Monte Carlo, Sieve, and Nbody). (4) The kinds of casts in translated programs could also affect the performance of `LearnPerf`. After checking the translated programs, we observed the presence of simple casts (about 63% of all casts) between basic types as well as higher-order casts (about 37% of all casts) between function types, list types, and object types. (5) Each time is an average of 10 runs and each machine learning experiment is averaged over 5 trials.

6 Related Work

Understanding performance changes during migration. While a lot of work has been done to automatically migrate dynamic programs toward more static, little work has been done to aid the performance aspect during program migration except for a few efforts.

Our work is closely related to the work by Campora et al. [8]. They developed Herder to help navigate the performance landscape during migration. However, there are many differences between `LearnPerf` and Herder. First, `LearnPerf` is able to precisely predict a time for any configuration while Herder is able to find only a symbolic overhead for each configuration. There is no direct mapping from these symbolic values to real runtimes

and so the relation between two symbolic values often does not carry over to the real runtimes. For example, two configurations from a single benchmark have symbolic values $2 * \ell_3 * \ell_4$ and $67 * \ell_3 * \ell_4$, respectively, while their corresponding runtimes are 24.79 and 37.38 seconds, respectively. As a result, several migration supports are possible with `LearnPerf` but not HERDER, such as classifying neighbors of a certain configuration based on their speedup/slowdown ratios.

Another difference is that, since our approach is based on machine learning, we only need to extract approximate values for features. HERDER, however, is based on static analysis and needs to be very accurate. For example, in `LearnPerf`, the overhead for a function cast is a simple addition of creation overhead and invocation overhead. In HERDER, a function cast needs to be transformed to an intermediate language to simulate the creation of proxies. As a result, it is easier to support more language features in `LearnPerf` than in HERDER. For example, we support object and class types, but they were missing in HERDER.

Greenman et al. [15] also investigated the performance problem during program migration but from a very different perspective. Through a large-scale empirical study, the authors studied how outputs from profilers may be exploited for proving migration supports. They considered seventeen strategies for how to avoid configurations with unacceptable performance and navigate to configurations to acceptable performance. Through the study, they generated three useful lessons for developers and one lesson for language designers for how to deal with the performance problem. Their focus is very different from our work in that we aim to predict the runtime for each configuration, and provide other migration supports, such as finding the best performing configuration among the neighbors, on top of that.

Assessing and Optimizing Gradual Typing Performance. Takikawa et al. [41] evaluated the performance of Typed Racket, focusing on the areas mixing untyped and typed code. The evaluation revealed significant runtime overhead in sound gradual typing. In evaluating the performance of a gradual type system, Greenman et al. [16] conducted a thorough analysis by fully annotating a series of benchmarks in Typed Racket. Absolute performance calculations were derived by generating a significant subset of configurations from the complete lattice of possible configurations. Performance ratios for each configuration were then compared against base configurations to identify K-step and D-deliverable configurations.

Since the report of the performance problem in gradual typing, a lot of work has been done to solve this problem, ranging from designing new type systems or new languages, inferring more types to reduce casts, to developing more efficient cast languages.

Rastogi et al. [34] introduced a type inference algorithm for existing gradually typed code, especially focusing on the inflow and outflow of types. Their approach supports open-world soundness to enable sound interactions with unseen code. Instead, Nguyen et al. [27] used static analysis to remove casts that always succeed without considering open-world soundness.

The idea of developing new languages to avoid expensive interactions has been explored by Muehlboeck et al. Muehlboeck and Tate [26]. Several approaches have been developed to exploit compilers or JITs to improve gradual typing performance Rastogi et al. [35], Richards et al. [36], Bauman et al. [5]. Another important line of improving gradual typing performance is through the design of new or change cast constructs Feltey et al. [12], Kuhlenschmidt et al. [20]. The work by Allende et al. [4] designed confined gradual typing, allowing users to control the flowing of type information through type annotations for reducing expensive boundary crossings.

Our approach is complementary to these approaches in that they do not try to compare the performance of different configurations and identify performant configurations. Also, while these approaches optimize the performance of gradual programs, they often do not

fully reduce the overheads due to runtime type checks. This paper shows that our approach works well for both the guarded and transient semantics. It looks promising in applying our idea to the translated programs from these approaches to predict the performance of these optimized programs.

Machine Learning for Programming. Many machine learning based approaches have been developed for solving programming language and software engineering problems Wan et al. [49], Allamanis et al. [2]. A main trend is using deep models, such as large language models, to automatically extract code features. Interestingly, our exploration shows that deep learning approach does not produce a good model for performance prediction for our problem. Vo and Nguyen [48] observed a similar phenomenon for vulnerability detection.

7 Conclusion

With gradual typing, developers enjoy the benefits of both static and dynamic typing. A major obstacle of adopting gradual typing is that the runtime overhead when going from less typed regions to more typed regions is often high and unpredictable. To address this issue, we developed a machine learning-based solution named `LearnPerf` that approximates runtime overheads due to inserted casts. We have evaluated our approach on 12 Python benchmarks, with each of the three large benchmarks having more than 1000 LOC. The evaluation results demonstrated that `LearnPerf` is able to precisely predict the execution time of each configuration. On top of that, we can develop further migration supports, such as finding the most performant neighbor of a configuration when it has poor performance. Our approach works well for both guarded and transient semantics. In the future, we would like to extend our approach to support a more macro level gradually-typed language, such as Typed Racket. It is also interesting to investigate if our approach can be employed to predict the performance of optimized gradual programs.

References

1 Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

2 Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), July 2018. doi:10.1145/3212695.

3 Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2020. doi:10.1145/3385412.3385997.

4 Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 251–270, New York, NY, USA, 2014. ACM. doi:10.1145/2660193.2660222.

5 Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound gradual typing: Only mostly dead. *Proc. ACM Program. Lang.*, 1(OOPSLA):54:1–54:24, October 2017. doi:10.1145/3133878.

6 John Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating gradual types. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '18, New York, NY, USA, 2018. ACM.

7 John Peter Campora and Sheng Chen. Taming type annotations in gradual typing. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428259.

8 John Peter Campora, Sheng Chen, and Eric Walkingshaw. Casts and costs: Harmonizing safety and performance in gradual typing. *Proc. ACM Program. Lang.*, 2(ICFP):98:1–98:30, July 2018. doi:10.1145/3236793.

9 Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. Gradual typing: A new perspective. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290329.

10 Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. Type inference for static compilation of javascript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 410–429, New York, NY, USA, 2016. ACM. doi:10.1145/2983990.2984017.

11 Fernando Cristiani and Peter Thiemann. Generation of typescript declaration files from javascript code. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2021, pages 97–112, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3475738.3480941.

12 Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. Collapsible contracts: Fixing a pathology of gradual typing. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276503.

13 Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 303–315, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676992.

14 Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *Proc. ACM Program. Lang.*, 2(ICFP):71:1–71:32, July 2018. doi:10.1145/3236766.

15 Ben Greenman, Matthias Felleisen, and Christos Dimoulas. How profilers can help navigate type migration. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. doi:10.1145/3622817.

16 Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. How to evaluate the performance of gradual type systems. *Journal of Functional Programming*, 29:e4, 2019. doi:10.1017/S0956796818000217.

17 Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint*, 2022. arXiv:2203.03850.

18 Sergiy Kolesnikov, Norbert Siegmund, Christian K´astner, Alexander Grebhahn, and Sven Apel. Tradeoffs in modeling performance of highly configurable software systems. *Software & Systems Modeling*, 18:2265–2283, June 2019. doi:10.1007/s10270-018-0662-9.

19 Erik Krogh Kristensen and Anders Møller. Type test scripts for typescript testing. *Proc. ACM Program. Lang.*, 1(OOPSLA):90:1–90:25, October 2017. doi:10.1145/3133914.

20 Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Toward efficient gradual typing for structural types via coercions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 517–532, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314627.

21 Andréa Matsunaga and José A.B. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504, 2010. doi:10.1109/CCGRID.2010.98.

22 Zeina Migeed and Jens Palsberg. What is decidable about gradual types? *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371097.

23 Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2241–2252, 2022.

24 Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. Dynamic type inference for gradual hindley–milner typing. *Proc. ACM Program. Lang.*, 3(POPL):18:1–18:29, January 2019. doi:10.1145/3290331.

25 Cameron Moy, Phúc C. Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. Corpse reviver: Sound and efficient gradual typing via contract verification. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434334.

26 Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. In *OOPSLA*, New York, NY, USA, 2017. ACM. doi:10.1145/3133880.

27 Phúc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification for higher-order stateful programs. *Proc. ACM Program. Lang.*, 2(POPL):51:1–51:30, December 2017. doi:10.1145/3158139.

28 Francisco Ortin, Miguel Garcia, and Seán McSweeney. Rule-based program specialization to optimize gradually typed code. *Knowledge-Based Systems*, 179:145–173, 2019. doi:10.1016/j.knosys.2019.05.013.

29 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

30 Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: A hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 2019–2030, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510003.3510038.

31 Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. Solver-based gradual type migration. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485488.

32 Marius-Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588, 2009.

33 Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. Typewriter: Neural type prediction with search-based validation, 2020. arXiv:1912.03768.

34 Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 481–494, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103714.

35 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *POPL*, 2015.

36 Gregor Richards, Ellen Arteca, and Alexi Turcotte. The vm already knew that: Leveraging compile-time knowledge to optimize gradual typing. *Proc. ACM Program. Lang.*, 1(OOPSLA):55:1–55:27, October 2017. doi:10.1145/3133879.

37 Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 17–31, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

38 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *In Scheme and Functional Programming Workshop*, pages 81–92, 2006.

39 Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 7:1–7:12, New York, NY, USA, 2008. ACM. doi:10.1145/1408681.1408688.

40 Dagmar Stumpfe and Jürgen Bajorath. Exploring activity cliffs in medicinal chemistry. *Journal of Medicinal Chemistry*, 55(7):2932–2942, 2012. PMID: 22236250. doi:10.1021/jm201706b.

41 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 456–468, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837630.

42 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 964–974, New York, NY, USA, 2006. ACM. doi:10.1145/1176617.1176755.

43 Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten Years Later. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SNAPL.2017.17.

44 Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. The behavior of gradual types: A user study. In *DLS*, number ICFP in DLS 2018, page 1–12, 2018. doi:10.1145/3393673.3276947.

45 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 45–56, New York, NY, USA, 2014. ACM. doi:10.1145/2661088.2661101.

46 Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. Optimizing and evaluating transient gradual typing. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2019, pages 28–41, New York, NY, USA, 2019. ACM. doi:10.1145/3359619.3359742.

47 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 762–774, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009849.

48 Hieu Dinh Vo and Son Nguyen. Can an old fashioned feature extraction and a light-weight model improve vulnerability type identification performance? *arXiv preprint*, 2023. arXiv:2306.14726.

49 Yao Wan, Yang He, Zhangqian Bi, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip S Yu. Deep learning for code intelligence: Survey, benchmark and toolkit. *arXiv preprint*, 2023. arXiv:2401.00288.

50 Jun Xia, Lecheng Zhang, Xiao Zhu, and Stan Z. Li. Why deep models often cannot beat non-deep counterparts on molecular property prediction? In *ICML 3rd Workshop on Interpretable Machine Learning in Healthcare (IMLH)*, 2023. URL: https://openreview.net/forum?id=hJG8xgj2Y5.

51 Ming-Ho Yee and Arjun Guha. Do machine learning models produce typescript types that type check? *arXiv preprint*, 2023. arXiv:2302.12163.