White-Box Program Tuning

Wen-Chuan Lee*, Yingqi Liu*, Peng Liu*, Shiqing Ma*, Hongjun Choi*, Xiangyu Zhang*, Rajiv Gupta‡
*Purdue University, USA
‡University of California, Riverside, USA

***Abstract*—Many programs or algorithms are largely parameterized, especially those based on heuristics. The quality of the results depends on the parameter setting. Different inputs often have different optimal settings. Program tuning is hence of great importance. Existing tuning techniques treat the program as a black-box and hence cannot leverage the internal program states to achieve better tuning. We propose a white-box tuning technique that is implemented as a library. The user can compose complex program tuning tasks by adding a small number of library calls to the original program and providing a few callback functions. Our experiments on 13 widely-used real-world programs show that our technique substantially improves data processing results and outperforms OpenTuner, the state-of-the-art black-box tuning technique.**

***Index Terms*—white-box tuning; black-box tuning; parameter tuning; parameterized program**

I. Introduction

More and more highly parameterized programs or algorithms are being used to solve different problems. Their complexity is also growing at an enormous pace, involving more and more computation stages. A prominent challenge for using these programs or algorithms is that the user has to configure a set of parameters beforehand. More importantly, the optimal configuration is mostly dependent on the specific input. Different inputs require different configurations to achieve the optimal results.

For instance, the results of *K-means* [46], a popular data clustering algorithm, heavily depends upon the choice of parameter K. It specifies the number of clusters into which the user wants to partition the input data. A lot of research [29, 53, 54, 73, 74] has aimed at automatically deriving the appropriate k value from the input. However, there is no general solution for finding K. Another example relates to object detection in satellite image processing [16]. The parametrized algorithm processes a large volume of images in a time unit to generate the detection results. However, the parameter configuration that yields the best results for one image may produce suboptimal results for another image (e.g., missing objects and broken edges). Consider, `Canny` [20], one of the most widely used image processing algorithms that detect edges. It is a multi-staged algorithm with three important parameters upon which `Canny`'s results heavily depend. According to [33], each input image may require a specific parameter setting to produce the best edge detection result. Fig. 1 shows the results on two different images using `Canny`. The left two are the original images. The other images show the results from two respective parameter configurations. Observe that configuration (0.6, 0.5, 0.9) produces the better result for the airplane whereas configuration (1.8, 0.2, 0.7) produces the better result for the trashcan. Thus, automated parameter tuning becomes critical in data processing as manual tuning is not realistic.

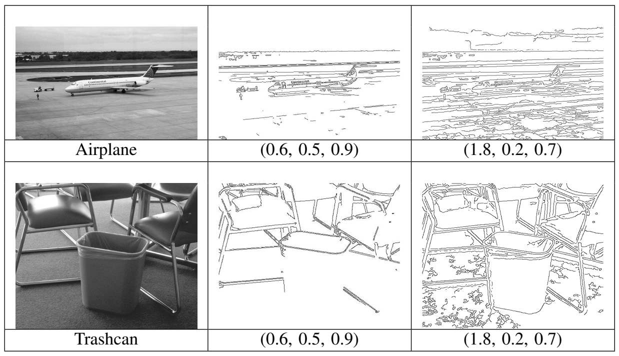

Fig. 1. `Canny`'s results with different parameters

A. Key Observation of Staged Computation Paradigm

By observing `Canny` and many other real world parameterized programs, we find that they typically follow the *staged computing paradigm*, i.e., they consist of multiple computation stages such that each stage has a unique set of tunable parameters.

B. Existing Work

Multiple frameworks were proposed to automate program tuning, among which OpenTuner [7] is the state-of-the-art. Oblivious of the staged computation paradigm, these frameworks treat the computation as a black-box. Guided by a user-provided scoring function of the final result, they sample the parameter space to find the best parameter configuration. Internally, they adopt stochastic algorithms [40, 64] or genetic algorithms [48] as the search strategy. While the above frameworks have achieved a certain level of success, they suffer greatly from poor performance due to the *inherent* limitations of the *black-box* designs:

- *All* parameters need to be tuned and set in each configuration, leading to an *exponential* number of configurations.
- A *full* execution accounts for the sampling of a *single* parameter configuration. Note that the full execution typically needs to load a large corpus of data and conduct lengthy preprocessing, which are very time consuming.

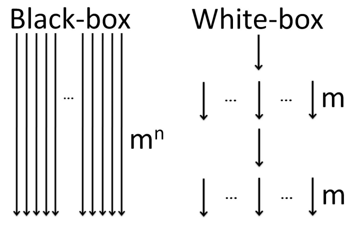

Fig. 2. Execution models of black-box and white-box tuning

C. Our Work

In this paper, we propose a novel white-box tuning framework called WBTuner. It is aware of the staged computation paradigm and tunes each stage independently. Specifically, WBTuner spawns multiple processes to sample different parameter configurations involved in a stage. At the end of each stage, it aggregates the sampled internal results of that stage through a default or custom aggregation strategy. The aggregation step reduces the spawned processes to fewer or one process (with desirable internal results achieved by tuning), which will proceed to tune the next stage. Intuitively, the aggregation strategy may either select the min/max value from the internal results (from various processes) or take he average value (Sec. IV-C).

Consider an application with n stages of computation, each stage having a unique parameter to tune. The parameter domain has m unique values. Initially, WBTuner spawns m sampling processes to cover the m configurations of the first stage. Assume only one of the processes in a stage is selected to proceed to the next stage (after aggregation). WBTuner needs only m sampling processes in each stage. Overall, WBTuner only needs to cover $m{*}n$ configurations and achieves so with a single full execution that keeps at most m live processes in any stage. Comparatively, OpenTuner needs to cover the m^n unique parameter configurations with m^n full execution instances. Fig. 2 illustrates the comparison.

D. Properties

WBTuner features the following properties.

- By leveraging the independence between stages, WBTuner needs to sample much fewer parameter configurations than OpenTuner. In the above example, it needs to sample only $m * n$ configurations, while OpenTuner needs to sample m^n configurations.
- Wasteful computation caused by poor internal results in an early stage can be terminated through the aggregation step for efficiency/efficacy. This is infeasible in black-box tuning.
- A full execution is reused for sampling different configurations and tuning different stages. Through the reused execution, WBTuner greatly reduces the number of full execution instances needed. Note that every full execution may need to load and pre-process a large corpus of data, which only have to be done once in WBTuner.

E. Contributions

The following shows our key contributions.

- We propose a novel white-box tuning technique. As discussed above, it features a set of salient properties compared to the existing black-box tuning.
- We develop a prototype WBTuner in the form of a library that offers the users flexible access to internal program states. The realization of the library incurs great challenges related to process management and data store management. Our technique addresses these problems through a novel runtime transparently to the end users. Besides, we formalize the semantics of the runtime execution.
- We release our implementation of WBTuner for the community at [71]. We use WBTuner to tune 12 widely used parameterized programs. Our experiments show that WBTuner substantially improves their results with reasonable overhead. The comparison with OpenTuner shows that OpenTuner takes 3.08X time to achieve the same results under a single core environment and 4.67X when multiple cores are used.
- We use WBTuner to tune the parameters of a complex drone controller software (278K LOC) to mimic the behavior of a different controller with a better configuration (Sec. V-B5). Changing the configurations manually is infeasible because the numbers of parameters are large (612 and 426 respectively) and the meanings of these parameters are quite different between the two controllers.

II. Overview of White-Box Tuning Framework

We present the interface and show how to use it to tune `Canny`, a popular image processing algorithm. Due to space limitation, detailed tuning examples and complete documentations can be found in [71].

A. User Interface

WBTuner provides the user with an intuitive interface, which consists of multiple tuning primitives shown in Fig. 3. They are essentially library calls in the same programming language as the original program, rather than annotations in some specification language. We use the following *Canny* example to intuitively explain how to use the interface.

$\mathit{Library\ Calls}:$	
	$@\mathbf{sampling}(n, \mathit{cbStrgy})\ \mid$
	$@\mathbf{aggregate}(x, \mathit{cbAggr})\ \mid$
	$@\mathbf{sample}(x, \mathit{cbDist})\ \mid\ @\mathbf{expose}(x)\ \mid$
	$@\mathbf{load}(x)\ \mid\ @\mathbf{loadS}(x, i)\ \mid\ @\mathbf{split}()\ \mid$
	$@\mathbf{sync}(\mathit{cbBarrier})\ \mid\ @\mathbf{check}(\mathit{cbChk})$
$\mathit{CallBack}:$	$\mathit{cbStrgy},\ \mathit{cbAggr},\ \mathit{cbDist},\ \mathit{cbChk},\ \mathit{cbBarrier}$

Fig. 3. Primitives

B. Running Example

`Canny` has four stages: the *Gaussian smoothing* stage (line 22 in Fig. 4) which removes noise from the input image, the *image transformation* stage (line 30) which performs non-maximal suppression, the *edge traversal* stage (line 37) which leverages the hysteresis analysis to track all potential edges in the image, and the *visualization* stage which visualizes the final results.

It takes three parameters: `sigma`, `low`, and `high`. Specifically, the Gaussian smoothing stage relies on the parameter `sigma` and the edge traversal stage relies on the `low` and `high` thresholds. Based on our observation, `Canny` is a representative of real world data processing applications, which usually follow the *staged computing paradigm*, i.e., they consist of multiple computation stages such that each stage has a unique set of tunable parameters.

Fig. 4 shows how the interface is used (symbol @ is replaced with *wbt_*). Primitive *wbt_sampling*() (line 20) denotes the start of a sampling code region. It specifies the number of samples that should be collected within this region and a callback function that implements a sampling strategy. WBTuner has a few built-in callbacks including `random` in this example. Primitive *wbt_aggregate*() (line 27) marks the end of a sampling region. It specifies a callback function (e.g., `AggregateGaussian()`) that aggregates the values of `sImage` across sample runs. Primitive *wbt_sample*() (line 21) indicates that a program variable, e.g., `sigma`, is a variable to tune (sample). It also specifies the distribution of the variable from which sample values are taken.

A callback function `AggregateGaussian()` is provided by the user to facilitate tuning. In this example, we implement it following an existing approach [39] to prune the poorly smoothed ones. Specifically, it loads (line 6) the images denoted by `sImage` which are computed according to different sampled values of `sigma` and determines (line 7) whether each image is properly smoothed given the image size `imgSize`. We will explain the relevant primitives `wbt_load()`, `wbt_loadS()` and `wbt_expose()` in Section III-A3. For each properly smoothed image, a new process is spawned by the primitive *wbt_split* (line 9) to continue to tune `low` and `high` in the edge traversal stage (lines 34-41), while preserving the sampled `sigma` value and the produced image, i.e., `sImage`. Next we will discuss the execution model that underlies the user interface.

```
 1 /* User provided callback */
 2 void AggregateGaussian() {
 3   for (int i=0; i<samplgNum; i++) {
 4     // Prune smdImage if over/under smoothed
 5     // Spawn a new process if not pruned
 6     result = wbt_loadS(sImage, i);
 7     if (properSmooth(result, wbt_load(imgSize))) {
 8       memcpy(sImage, result, wbt_load(imgSize));
 9       wbt_split();
10     }
11   }
12 }
13 /* Source program */
14 void canny() {
15   // Initializations
16   image = input(file);
17
18   /* STAGE ONE */
19   // Begin uniform sampling in 0.1 <= sigma <= 10
20   wbt_sampling(600, random);
21   sigma = wbt_sample(uniform(0.1,10));
22   sImage = gaussianSmooth(image, sigma);
23
24   //End sampling and Spawn new processes for those
25   //samples with appropriately smoothed images
26   wbt_expose(imgSize);
27   wbt_aggregate(sImage, AggregateGaussian);
28
29   /* STAGE TWO */
30   // Gradient computation and suppression
31
32   /* STAGE THREE */
33   // Begin sampling, 0.1 <= low, high <= 1
34   wbt_sampling(20, random);
35   low = wbt_sample(uniform(0.1,1));
36   high = wbt_sample(uniform(0.1,1));
37   finalImage = hysteresis(low, high);
38
39   // End sampling and aggregate finalImage from
40   // each sample to do majority vote
41   wbt_aggregate(finalImage, MajorityVote);
42
43   /* STAGE FOUR */
44   visualize(finalImage);
45 }
```

Fig. 4. White-box tuning for `Canny`. The highlighted statements are added. Tuning primitives start with `wbt`.

C. Runtime Execution Model

The runtime execution framework is shown in Fig. 5. Initially, the original main process executes normally until it reaches the start of a tuning region (①). At this point, its role is switched to a *tuning process*. Intuitively, a tuning process is the "manager" of a pool of *sampling processes* that it spawns. A sampling process is the "worker" that conducts the computation within the region, and emits its result at the end of the region. The tuning process invokes the *sampling driver* (②) to spawn a pool of child sampling processes (③). The driver determines how many sampling processes to be spawned and exercises a given sampling strategy. In some cases, the sampling strategy is feedback driven and relies on previous tuning results.

After spawning, the tuning process pauses. The sampling processes carry out the computation within the tuning code region (④), orchestrated by a scheduler (Sec. III-B2). When a sampling process encounters a *tuning variable* (i.e., X), it acquires a sample value from the variable's distribution. The sampling processes have different states afterwards. Upon reaching the end of the tuning region, a sampling process calls the *child aggregation driver* (⑤) to commit its own computation result from the *sample result variable* (i.e., Y) and terminates. Note that although the sampling process also calls the primitive *wbt_aggregate*(), it only submits its sampling outcome. After all sampling processes commit, the tuning process resumes and invokes the *parent aggregation driver* to aggregate the sampling results (⑥). It then continues to execute normally with the aggregated results (⑦).

The above simplified model assumes a single tuning procecss in the runtime system. It is usually necessary to have multiple tuning processes. For example, consider the aggregation at line 27 in Fig. 4, the user may want to spawn multiple (independent) tuning processes each continuing with one from a subset of good internal results, i.e., properly smoothed images referred to by `sImage`, rather than a single tuning process that continues with exactly one internal result. To

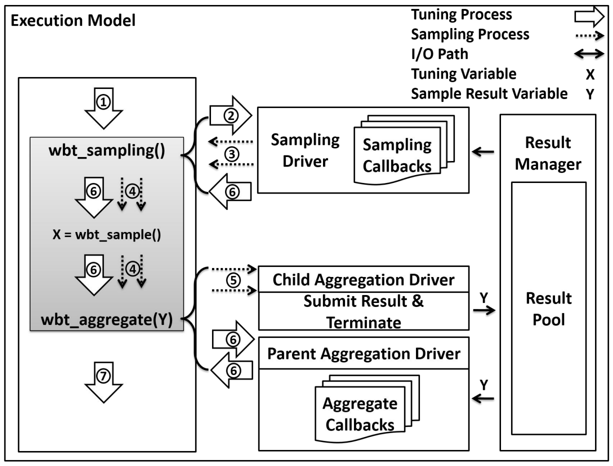

Fig. 5. Execution Model

achieve this, the user can use primitive $wbt\_split()$ (line 9) to explicitly spawn a new tuning process (not sampling process) if the image is properly smoothed (line 7). Our runtime system fully supports multiple tuning processes (Section III-A2).

D. Result and Comparison

Initially we use 200 samples (line 20). At the end of the Gaussian smoothing stage (line 27), the invoked function `AggregateGaussian()` prunes 78 samples that are not properly smoothed, and keeps 122 samples. WBTuner further spawns a tuning process for each remaining sample. When each of these processes reaches the edge traversal stage (line 34), it triggers a new sampling procedure which explores 90 samples(with different configurations of the parameters `low` and `high`) for each smoothed image. Hence the total number of samples is 122×90=10980. Fig. 6 shows the tuning model of WBTuner for `Canny`.

The sampling results are aggregated by majority voting (line 41), that is, a pixel is set if it is set in the majority of sample runs. WBTuner supports voting by default. Hence, the user can aggregate results through one line of function call. Finally, the aggregated image is visualized at line 44.

For comparison, we also apply OpenTuner to tune `Canny` with its default search strategy (i.e., Multi-armed bandit). Since no algorithm exists for computing a score for the output quality, we use simple heuristics to determine the poor samples, such as those that have very few or too many pixels in the final image. We use the execution time of WBTuner as the timeout for OpenTuner. The images generated by OpenTuner through its sampling runs are aggregated by the same voting procedure in WBTuner.

The tuning results for the coffeemaker image are shown in Fig. 7. Observe that WBTuner spent 90 seconds on 9040 samples whereas OpenTuner can only finish 842 samples within the same amount of time, because most of its computation time was spent on the expensive image loading, Gaussian smoothing, and gradient computation stages as it has to repeat such computation for each sample run. In addition to the visual result, we use the SSIM score [70] to compare the result with the ground truth result hand-picked by experts [33]. Both visual and scoring results demonstrate that that WBTuner outperforms OpenTuner.

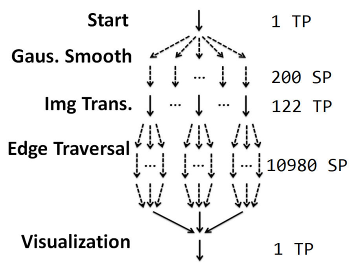

Fig. 6. Tuning Canny. TP/SP are tuning/sampling processes.

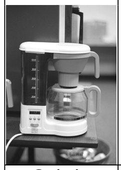	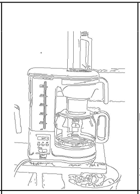	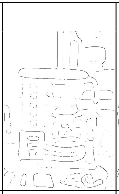	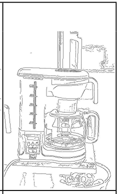
Origin	Ground Truth	OpenTuner	WBTuner
samples	-	842	10980
SSIM	1	0.592	0.794

Fig. 7. Tuning Canny with image coffeemaker in 90s.

III. Execution Model: Semantics and System

In order to achieve white-box tuning, we need to overcome a number of prominent challenges related to the management of *stores* and *processes*. First, an original process will spawn many sampling processes, which may need to be terminated (if the sampling result is poor), communicate with each other, further spawn their own child sampling processes, and join at specific execution points. Second, as the sampling processes produce a lot of sample data from internal states, managing such data (i.e., storing, accessing, and aggregating results across processes) is also challenging. All these complexities should be transparent to the users. In Section III-A, we describe the formal execution model of our runtime system with the operational semantics. In Section III-B, we present the implementation details of our runtime system.

A. Semantics

The semantics are presented in Fig. 8. The related definitions are presented at the top of the figure.

1) Stores: WBTuner has two stores, the *store* σ for original program states and the *sample store* δ that is shared across all processes to store sampling outputs. The two are isolated. State transferring between the two are performed explicitly by the programmer. In δ, states can be further divided into two classes: (1) exposed store, a store for exposed variables, (2) aggregation store, a store for sampled results from the child sampling processes.

DEFINITIONS: $Store\ \sigma ::= Var \rightarrow Value \quad SmpStore\ \delta ::= Var \rightarrow Value \mid Var \rightarrow (Index \rightarrow Value) \quad Mode\ \omega ::= \mathbb{T}\langle pid \rangle \mid \mathbb{S}\langle pid \rangle$
$Stmt\ s ::= ... \mid \mathbf{spawn}(\sigma, \delta, \omega, s) \mid \mathbf{notify}(pid) \mid \mathbf{wait}(pid) \mid \mathbf{invoke}(cb)$

STATEMENT RULES: $\boxed{\sigma, \delta, \omega : s \xrightarrow{s} \sigma', \delta', \omega', s'}$ *Let* $CPID = \{Child\ Process\ ID\}$, $PPID = Parent\ Process\ ID$ in the following rules:

$\sigma,\ \delta,\ \omega:\ x := v$	$\xrightarrow{s}$	$\sigma[x \mapsto v],\ \delta,\ \omega,\ \mathbf{skip}$	$[ASSIGN]$
$\sigma,\ \delta,\ \mathbb{T}\langle pid \rangle:\ @\mathbf{sampling}(n, cbStrgy);\ s$	$\xrightarrow{s}$	$\sigma,\ \delta,\ \mathbb{T}\langle pid \rangle,\ \forall i \in [1,n],\ \mathbf{spawn}(\sigma,\ \delta,\ \mathbb{S}\langle i \rangle,\ \mathbf{invoke}(cbStrgy); s);\ \mathbf{invoke}(cbStrgy);\ s$	$[SAMPLING]$
$\sigma,\ \delta,\ \mathbb{T}\langle pid \rangle:\ @\mathbf{aggregate}(x, cbAggr);\ s$	$\xrightarrow{s}$	$\sigma,\ \delta,\ \mathbb{T}\langle pid \rangle,\ \mathbf{invoke}(cbAggr, x);\ s$	$[AGGR-T]$
$\sigma,\ \delta,\ \mathbb{S}\langle pid \rangle:\ @\mathbf{aggregate}(x, cbAggr);\ s$	$\xrightarrow{s}$	$\sigma,\ \delta[x[pid] \mapsto \sigma(x)],\ \mathbb{S}\langle pid \rangle,\ \mathbf{skip}$	$[AGGR-S]$
$\sigma,\ \delta,\ \mathbb{S}\langle pid \rangle:\ @\mathbf{sample}(x, cbDist);\ s$	$\xrightarrow{s}$	$\sigma,\ \delta,\ \mathbb{S}\langle pid \rangle,\ x := \mathbf{invoke}(cbDist);\ s$	$[SAMPLE]$
$\sigma,\ \delta,\ \mathbb{T}\langle pid \rangle:\ @\mathbf{split}();\ s$	$\xrightarrow{s}$	$\sigma,\ \delta,\ \mathbb{T}\langle pid \rangle,\ \mathbf{spawn}(\sigma, \{\}, \mathbb{T}\langle \mathbf{newPid}() \rangle, s);\ s$	$[SPLIT]$
$\sigma,\ \delta,\ \mathbb{T}\langle pid \rangle:\ @\mathbf{sync}(cbBarrier);\ s$	$\xrightarrow{s}$	$\sigma,\ \delta,\ \mathbb{T}\langle pid \rangle,\ \forall i \in CPID,\ \mathbf{wait}(i);\ \mathbf{invoke}(cbBarrier);\ \forall i \in CPID,\ \mathbf{notify}(i);\ s$	$[SYNC-T]$
$\sigma,\ \delta,\ \mathbb{S}\langle pid \rangle:\ @\mathbf{sync}(cbBarrier);\ s$	$\xrightarrow{s}$	$\sigma,\ \delta \mapsto \sigma(x)],\ \mathbb{S}\langle pid \rangle,\ \mathbf{notify}(PPID),\ \mathbf{wait}(PPID);\ s$	$[SYNC-S]$
$\sigma,\ \delta,\ \mathbb{S}\langle pid \rangle:\ @\mathbf{check}(cbChk);\ s$	$\xrightarrow{s}$	$\sigma,\ \delta,\ \mathbb{S}\langle pid \rangle,\ \mathbf{if}\ \mathbf{invoke}(cbChk) \equiv true\ \mathbf{then}\ s\ \mathbf{else}\ \mathbf{skip}$	$[CHECK]$
$\sigma,\ \delta,\ \mathbb{T}\langle pid \rangle:\ @\mathbf{expose}(x);\ s$	$\xrightarrow{s}$	$\sigma,\ \delta[x \mapsto \sigma(x)],\ \mathbb{T}\langle pid \rangle,\ s$	$[EXPOSE]$
$\sigma,\ \delta,\ \mathbb{T}\langle pid \rangle:\ y = @\mathbf{load}(x);\ s$	$\xrightarrow{s}$	$\sigma[y \mapsto \delta(x)],\ \delta,\ \mathbb{T}\langle pid \rangle,\ s$	$[LOAD]$
$\sigma,\ \delta,\ \mathbb{T}\langle pid \rangle:\ y = @\mathbf{loadS}(x, i);\ s$	$\xrightarrow{s}$	$\sigma[y \mapsto \delta(x)[i]],\ \delta,\ \mathbb{T}\langle pid \rangle,\ s$	$[LOADSAMPLE]$

Fig. 8. Operational Semantics

a) Exposed Store: Exposed store is a mapping from variables to values. A local variable is exposed by the primitive $wbt\_expose()$. The exposed local variable is saved to the exposed store and can be retrieved with the primitive $wbt\_load()$. Different from common local variables, the exposed local variable is available outside its local scope (e.g., function). Thus, the exposed local variable can be used to pass the value across different scopes. For instance, in Fig. 4, the local variable `imgSize` from the `canny` function is exposed at line 26 and then loaded at line 7 in the `AggregateGaussian` function.

b) Aggregation Store: Aggregation store of a tuning process stores the sampled outcomes. It maps each program variable x to a vector $\delta(x)$, of which the ith entry holds the value of the variable from the ith child process. Note that vector abstracts the mapping from index to values. At the semantic level, the primitive $wbt\_aggregate(x, \dots)$ forces each child sampling process to write/commit the value of x from its regular store to the aggregation store of the parent tuning process, as illustrated by line 27 in Fig. 4. The primitive $wbt\_loadS(x, i)$ loads the value of x from the ith child process, as illustrated by line 6 in Fig. 4.

2) Processes: WBTuner supports two execution modes, $\mathbb{T}\langle pid \rangle$ denotes the a tuning process and $\mathbb{S}\langle pid \rangle$ is a sampling process. pid denotes the process id. To facilitate discussion, we extend the statements to include a $\mathbf{spawn}(\sigma, \delta, \omega, s)$ statement that forks a process with the specified stores, execution mode, and the process body s, a $\mathbf{notify}(pid)$ statement that notifies a process pid, a $\mathbf{wait}(pid)$ statement that waits for a notification from the process pid, and an $\mathbf{invoke}(\mathbf{cb})$ statement that invokes a callback function cb.

3) Statement Rules: Rule [*SAMPLING*] forks n sampling processes (indicated by the $\mathbb{S}\langle i \rangle$ mode) through the $\mathbf{spawn}()$ primitive. Observe that the last parameter of the primitive is the body of the child process, which contains the same statements as the parent, namely, "$\mathbf{invoke}(cbStrgy); s$". After forking, callback $cbStrgy()$ is called to initialize the sampling strategy in the children. Note that Rule [*SAMPLING*] only applies in a tuning process. It is a NOP in a sampling process.

Rule [*AGGR-T*] specifies that a tuning process invokes the callback $cbAggr()$ to aggregate the sampling results for variable x. In the callback, the user can implement various aggregation strategies. For example, the values of sample target variable x from all sample processes can be averaged and written back to x in the tuning process, which can proceed with the aggregated value. In contrast, Rule [*AGGR-S*] specifies that upon aggregation, a sampling process stores its sampling outcome of x to the element of the sampling vector corresponding to the process id and then terminates. Recall that only the tuning process aggregates results and sampling processes only produce results.

Rule [*SAMPLE*] only applies to sampling processes. It specifies that the callback $cbDist()$ is invoked to acquire a sample value for variable x, which denotes a parameter to tune. Rule [*SPLIT*] specifies that a tuning process can explicitly spawn a child tuning process. The child process is for tuning the next phase. Function $\mathbf{newPid}()$ returns a new pid. The child process inherits the regular store but not the sample store from the parent. Rule [*SYNC-T*] indicates that the tuning process waits for all the child sampling processes to reach the barrier, and then it invokes $cbBarrier()$ to perform some operations that access results across multiple sample runs. After that, the tuning process notifies all its child sampling processes to proceed. Compared to $@aggregate$, $@sync$ is usually used in the middle of a sampling region. Rule [*SYNC-S*] specifies that a sampling process notifies its parent tuning process after it has reached the barrier. It then waits for the tuning process to finish the callback and notify it to proceed. Notifications from child processes are queued to avoid message lost which may lead to deadlocks.

Rule [*CHECK*] specifies that a sampling process invokes a callback $cbChk()$ to check its local states. If the check returns $false$, the sampling process is terminated. This feature allows us to terminate useless sample runs long before they get to the aggregation point (e.g., `k-means` in Sec. V-B3), which improves not only the performance but also the final results. Note that such improvements are impossible to achieve in black-box tuning.

Rule [*EXPOSE*] exposes the value of x from the regular store to the sample store, which is accessed by tuning callbacks. The rule only applies to tuning processes. Observe that it allows callbacks to access program variables outside their scopes. Rule [*LOAD*] loads an exposed variable x (from the exposed store of δ) inside some callback function in a tuning process. Rule [*LOADSAMPLE*] loads the ith sample outcome of x (from the aggregation store of δ).

B. WBTuner Runtime System

We present the implementation details of WBTuner runtime by following the same structure as Section III-A.

1) Stores:

a) Exposed Store: We implemented the exposed store as follows. Our system encodes a local variable with its name and its scope information (e.g., the function name) before mapping it to the value in the exposed store. Similarly, our system uses the name and the scope information of a variable to retrieve the associated value. The encoding guarantees we can access the value of the exposed variable throughout the whole execution. Note that the scope information is required to distinguish the local variables with the same name from different scopes.

b) Aggregation Store: Our system achieves the semantics by leveraging the file system in disk. In particular, all sampled outcomes (WBTuner supports multiple sample result variables aggregation) of a sampling process are stored in a file. The file name is in the form pid, which specifies the sampling process that submits the results of the variables. All the files are stored in a directory owned by the tuning process. To load the ith outcome of x from disk, our system searches in the directory owned by the tuning process for the related file based on the information in primitive $wbt\_loadS(x, i)$.

2) Processes:

a) Process Scheduling: In practice there will be large number of tuning and sampling processes executing concurrently at runtime. Thus, WBTuner provides a scheduler to manage the creation and termination of processes. It prevents excessive process creation for better performance (Fig. 10). Using a uniform process pool is not optimal because of the difference between the two kinds of processes (tuning and sampling). Instead, we prioritize a sampling process over a tuning process because the former conducts the real computation. In addition, we want to finish all the sampling processes belonging to a tuning process as soon as possible so that the tuning process can finish its work and yield the resource.

The scheduler works as follows. Upon a spawn request, it checks if there are enough resources. If not, the current process is put in a priority queue. Upon a process termination event, the highest priority process in the queue is woken up.

Algorithm 1 shows the details. The $Schedule$ procedure is called with pid (i.e., process id), $event$ and $todo$. There are three possible events: $SPAWN\_S$ (i.e., spawning a sampling process), $SPAWN\_T$ (i.e., spawning a tuning process), and $EXIT$. The parameter $todo$ denotes the number of samples remained for the current (tuning) process. Sampling processes are ordered inside the priority queue based on the $todo$ values of their parent tuning processes. Lines 2-7 correspond to process termination, which wakes up the process with the highest priority. At line 8, the computed threshold denotes the remaining resources. If the number of available resources is below it, the current process will be put back into the priority queue (lines 9-12). Otherwise, it is allowed to proceed (line 14). Since real tuning is done by sampling processes, the threshold is always 0 for sampling processes so that they don't have to wait if there is any available resource. A configurable variable is used to prevent spawning too many tuning processes because they would inevitably lead to decreasing the tuning efficiency. In Algorithm 1, we set the configurable threshold of tuning process to 75% (i.e., it has to wait if 25% processes are occupied).

For benchmarks requiring a large number of samples and consuming lots of memory (e.g., `Canny`), the scheduler limits the number of concurrent samples and reduces the memory consumption and execution time significantly (Fig. 10). Too much memory consumption will result in excessive page fault which degrades the runtime performance.

IV. Practical Challenges

A. Overfitting

Since machine learning algorithms normally produce models as their output, the tuning task of these hyper-parameters is usually guided by the execution results of the models (e.g., lower classification errors). However, it might lead to *overfitting*, meaning that the model with the tuned hyper-parameters produces optimal results with training data but poor results with testing data. Note that programs such as `Canny` do not have this problem as they are tuning for the final result but not models tested by new data. Cross-validation can help to mitigate such hyper-parameters tuning problem according to existing studies [37, 50]. Specifically, it searches for the model hyper-parameters that generalize, rather than fitting to the training dataset. WBTuner provides intrinsic support to address overfitting by combining its execution model with *k-fold cross-validation* [65], a widely used technique. Specifically, to tune the parameters in a machine learning algorithm, the user only indicates the k value in the $wbt\_sampling()$ primitive and provides a validation callback. WBTuner will then transparently include k-fold cross validation during its tuning process. The experimental results in Fig. 17 demonstrate the necessity of cross-validation.

The tuning-validation model is shown in Fig. 9. First, the input data is transparently divided to k datasets for each sample run. Moreover, for the xth original sample run, WBTuner spawns $k - 1$ more processes, that form a *sampling and validation group* (SVG). If the user intends to collect n samples originally, WBTuner internally creates n SVGs, that is, $n*k$ processes. All the k processes in a SVG share the same sample values for the tuning variables but use different datasets for training and validation to prevent overfitting. As illustrated in the figure, the ith process in the SVG uses the ith dataset for validation and the remaining $k-1$ datasets for training. At the end of the execution of an SVG process, WBTuner invokes the user-supplied validation callback to apply the produced model

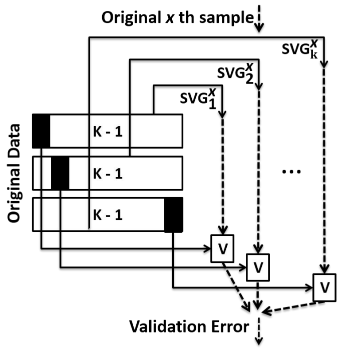

Fig. 9. Tuning + Validation Execution Model

on its validation dataset and computes the validation error. The validation errors from all SVG processes are then aggregated to drive the remaining steps of the tuning procedure. The experimental results in Fig. 17 demonstrate the necessity of cross-validation.

B. Incremental Aggregation

According to the execution model of WBTuner, the sampling results are submitted by the sampling processes and aggregated by the tuning processes once the sampling is completed. However, the whole execution entails massive storage and I/O overhead. According to our observation, aggregation can be performed incrementally in many benchmark programs as their aggregations involve functions such as finding the min, max, average, or majority. For instance, for the aggregation strategy min/max, each sampling process updates a shared global variable min/max by comparing its local outcome with the global variable. For incremental averaging, WBTuner uses a shared ring buffer to which sampling processes copy their results. The tuning process then consumes the data from the buffer to perform incremental averaging. Majority voting is handled in a similar fashion. In our experiment section (Fig. 10), we will show that incremental aggregation substantially reduces the tuning time and memory consumption.

C. Sampling/Aggregation Strategies

In addition to custom strategies provided by the user, WBTuner supports several common sampling/aggregation strategies by default. The user only needs to indicate the strategy name inside the $wbt\_sampling$/$wbt\_aggregate$ primitive to use it. Currently, the supported sampling strategies are random (RAND) and Markov Chain Monte Carlo (MCMC). For aggregation strategies, WBTuner supports min, max, majority vote (MV), averaging (AVG), and duplicate elimination (DEDUP). Based on our experience, these strategies are usually sufficient for most of the tuning tasks. Observe that only four benchmarks (out of 13 benchmarks) use custom aggregation strategies in our experiments.

D. Auto-tuning Sampling Number

Since the number of samples varies from one tuning region to another, WBTuner provides an automatic way similar to exponential backoff [12] to determine the optimal number of samples. For the sampling number of each primitive $wbt\_sampling()$, WBTuner doubles it and compares the aggregated results between the samples from original set and the samples from doubled set according to the scoring function. If the doubled result is better, the number of samples is doubled again until the no further improvements.

Algorithm 1 Process Scheduling

```
1: procedure SCHEDULE($pid, event, todo$)
2:     if $event$ = EXIT then
3:         $poolSize \leftarrow poolSize + 1$
4:         if $PQueue$ not EMPTY then
5:             $p \leftarrow PopPQueue()$
6:             $signal(p.pid)$
7:         return
8:     $threshold \leftarrow (event =$ SPAWN_S) ?
           $0 : MAX\_POOL\_SIZE \times 0.75$
9:     while $poolSize <= threshold$ do
10:        $PushPQueue($new $P(pid, event, todo))$
11:        $poolSize \leftarrow poolSize + 1$
12:        $wait()$
13:        $poolSize \leftarrow poolSize - 1$
14:    $poolSize \leftarrow poolSize - 1$
15:    return
```

V. Evaluation

WBTuner is implemented in C and publicly available at [71]. We evaluate the efficiency and effectiveness of WBTuner and compare it with OpenTuner. Experiments were run on a machine with Intel i7-2640M 2.80GHz processor and 16GB RAM.

Benchmarks. We use a wide variety of C/C++ benchmarks in our experiments, including 12 widely used data processing programs and a complex open-source controller software for commercial drones. These are heavily parameterized applications. For more benchmark information, please refer to the supplementary material [71] Section 2.

All programs have multiple datasets that can be found online or come with the program. We have selected only the datasets that have the outcome ground truth for comparison. On average, we used 10 datasets for each program. The results are summarized in Table I. Benchmarks either come with their own scoring functions or use publicly available scoring functions, so the callbacks for them are implemented accordingly. Comparison results of benchmarks without scoring functions (i.e., with superscript 1 in Tab. I) are explained in section V-A.

Columns 1-2 show the programs names and the lines of code. Column 3 shows the number of tunable parameters and column 4 shows the number of WBTuner primitives added to the source. The next two columns (5-6) describe the sampling and the aggregation strategies. Most programs use random sampling. `DBScan` and `K-means` demonstrate the use of a different sampling strategy (MCMC). `C4.5` and `SVM` use random sampling together with cross-validation, which is also implemented in OpenTuner for these two benchmarks (for

TABLE I
BENCHMARK STATISTICS AND THE EXPERIMENT RESULTS FOR ACHIEVING THE BEST TUNING SCORES.

Program	LOC	#P	#PR	Sampling	Aggregation	Ext LOC	Single Core							Multi Core					
							Native		WBTuner		OpenTuner		o/h(x)	Native	WBTuner		OpenTuner		o/h(x)
							time(s)	Score	time(s)	Score	time(s)	Score	OT/WB	time(s)	time(s)	Score	time(s)	Score	OT/WB
[20] ↑ Canny [1]	1.1k	3	8	RAND	CUSTOM/MV	151	0.159	0.29	51.53	0.636	t/o[2]	0.44	-	0.061	17.75	0.636	t/o	0.44	-
[17] ↑ Watershed [1]	270k	3	5	RAND	MV	34	1.03	0.41	26.1	0.65	31.5	0.65	2.11	0.93	56.1	0.65	221.59	0.65	3.95
[46] ↑ Kmeans	1.2k	1	5	MCMC	MAX	56	0.165	0.46	1.57	0.523	9.7	0.523	5.79	0.057	0.56	0.523	2.49	0.523	4.45
[28] ↑ DBScan	908	2	7	MCMC	MAX	80	0.657	0.299	25.41	0.502	124.08	0.502	3.21	0.021	2.94	0.502	15.7	0.502	5.34
[18] ↓ Face Rec	9.6k	3	7	RAND	MIN	92	4.788	17	578.62	7.3	1203.25	7.3	2.07	4.6	33.47	7.3	684.12	7.3	4.44
[42] ↑ Speech Rec [1]	19.8k	16	18	RAND	MV	89	4.263	1	313.25	5	t/o	4.2	-	4.12	19.54	5	t/o	4.2	-
[58] ↓ Phylip	12.6k	4	12	RAND	DEDUP/MIN	95	4.67	20.4	1021.4	0.84	1910.23	0.84	1.87	2.4	211.15	0.84	693.21	0.84	3.28
[57] ↑ FASTA	77.5k	2	4	RAND	CUSTOM	108	0.12	40	1.56	523	4.91	523	3.54	0.02	0.25	523	t/o	461	-
[55] ↑ TOPN Rec	33.5k	3	5	RAND	MAX	3	6.16	0.1	273.45	0.126	560.5	0.126	3.04	5.9	81.2	0.126	513.1	0.126	6.32
[38] ↓ METIS	44.3k	3	5	RAND	MAX	30	0.16	6952	4.77	6706	20.57	6706	4.31	0.06	1.2	6706	7.34	6717.7	6.12
[60] ↓ C4.5	17.8k	2	4	RAND+CV	MIN	58	0.059	2.46	7.23	0.082	21.54	0.082	3.18	0.036	1.68	0.082	6.54	0.082	3.89
[25] ↓ SVM	11.3k	8	10	RAND+CV	MIN	44	6.172	87	233.72	9.5	438.12	9.5	1.96	5.314	66.98	9.5	288.23	9.5	4.3
[45] ↓ Ardupilot	278k	40	44	RAND	CUSTOM	204	-	1954k	-	-	-	-	-	192.3	151k	1074k	-	-	-

↑: Higher scores are better; ↓: lower scores are better. 1. These benchmarks do not have default scoring functions.
2. "t/o" means OpenTuner cannot achieve the similar score (i.e., difference $<$ 10%) of WBTuner.

comparison). Column 7 presents the LOC in tuning callback functions. Observe that the number of primitives is small, yet, it allows to represent complex tuning models as we will demonstrate in Section V-B. The LOCs for callbacks are small compared to the source code LOCs. They mainly implement scoring functions or checks.

A. Tuning Results Summary

In the first experiment, we ran each benchmark with the largest dataset under three settings – (1) native run without tuning; (2) white-box tuning with WBTuner; (3) black-box tuning with OpenTuner and its default search strategy (i.e., multi-armed bandit [30]).

We ran WBTuner with the number of samples auto-tuned (Sec. IV-D) by WBTuner until converging, then we collected the tuning time. For OpenTuner, we gradually increased the timeout parameter until it either reaches similar results as WBTuner (difference $<$ 10%) or could not reach similar results after spending 10 times WBTuner 's tuning time. We measured the quality of the tuning results by comparing with the ground truth that comes with the datasets. Note that these ground truths are only used in measuring quality, *but not in tuning*. As stated before, OpenTuner requires scoring functions to guide the search; however, a few benchmarks do not have a standard scoring function (marked with the superscript 1 in Table I). To achieve fair comparison, for these benchmarks, we implemented the same domain-specific heuristics from WBTuner in OpenTuner to distinguish good and bad samples, and to use the same aggregation method from WBTuner to aggregate the good sample results. To quantify the results for these programs, we compute their scores based on the comparison with the ground truths. Such scores are not used in tuning.

Since OpenTuner does not support parallel sampling by default, which requires substantial engineering effort, we conducted the comparison in both single-core and multi-core. The single-core results are shown in columns 8-14 in Table I, while the multi-core results are shown in columns 15-20. Columns 8 and 9 present the native execution time and the score without tuning. Note that for the programs with ↑, the higher the scores the better, and for the others with ↓, the lower the scores the better. Column 10 presents the tuning time of WBTuner upon convergence. Column 11 shows the converged score. Column 12 shows the tuning time for OpenTuner. Those with "`t/o`" mean that those scores are apparently worse (difference $>$ 10%) than WBTuner after spending 10x more tuning time. Column 13 shows the final tuning score of OpenTuner. Column 14 shows the overhead comparison. Columns 15-20 are the results for multi-core.

Observe that for single-core environment, OpenTuner times out in 2 out of the 13 cases. For the other cases, the average tuning overhead of OpenTuner is 3.08X higher than WBTuner. For multi-core environment, 3 cases time out and the overhead ratio is 4.67X.

Observe that WBTuner substantially improves the results quality compared to those without any tuning. It is more effective than OpenTuner. For the cases that OpenTuner can reach the scores of WBTuner, we also allow more tuning time; still, it did not produce better results.

Fig. 10 shows the effect of the optimizations discussed in Sec. III-B2 and IV-B. Observe that the incremental aggregation is highly effective for several cases, especially for reducing the memory usage as it prevents reading a large number of results for one-shot aggregation. Observe that the scheduler further improves the performance in several cases, especially Canny and K-means. Before optimization, Canny's execution time and memory overhead are about 4X higher.

B. Tuning Case Studies

In this section, we study the details of tuning several representative programs under the single core environment and tuning Ardupilot (a drone controller) in the multi-core environment. Part of the discussion is moved to the supplementary material [71] Section 2.1 due to the space limitation.

1) Image Processing:

Canny. In Section II, we have already shown the tuning results of `Canny`. Here we used 10 different images from [33], where

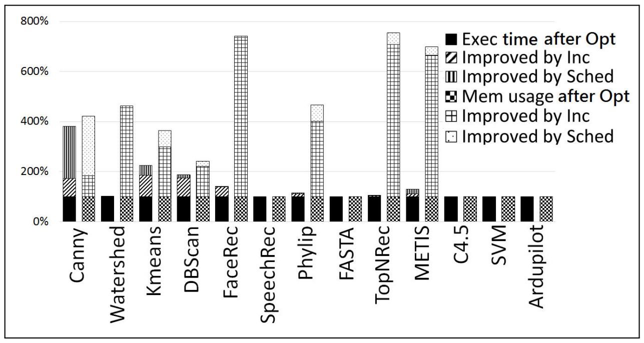

Fig. 10. Optimization effects on different benchmarks

each image has a ground truth result image hand-picked by experts.

Since no general scoring function exists, we use majority vote for results aggregation, meaning the result with the largest number of supports from the sample runs is reported. Then we use the SSIM [70] score to compare the voting result with the ground truth. The higher the score the better. We extended OpenTuner with the majority voting capability to achieve fair comparison. For each image, we ran WBTuner and OpenTuner 10 times and took the average. Fig. 11 shows the tuning score when WBTuner converges, the corresponding OpenTuner score after it runs the same amount of time, and the score without tuning. Observe that WBTuner almost always produces the best results.

On average, OpenTuner has 119% improvement over no-tuning, whereas the improvement of WBTuner is 178%. The reason is that WBTuner can prune a lot of sample runs that will not yield promising results after stage one (Fig. 4).

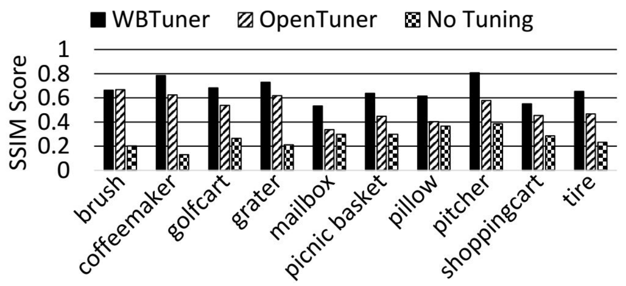

Fig. 11. `Canny` tuning scores of 10 images.

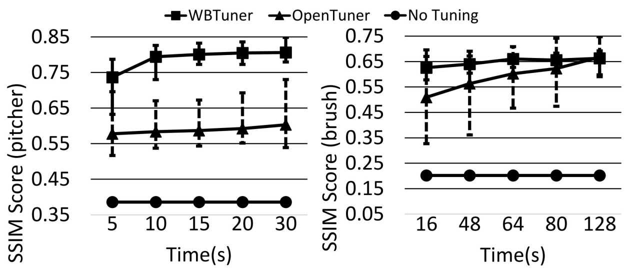

Fig. 12. `Canny` tuning score variation

The score variation with the tuning time is shown in Fig. 12 for the `pitcher` and `brush` images, which represent the maximum and minimum improvement over OpenTuner, respectively. Observe that for `pitcher`, even 5-second tuning in WBTuner yields much better results for 30 seconds tuning in OpenTuner. The visualization in Fig. 13 shows that the result by WBTuner is very close to the ground truth but the result by OpenTuner is not. For `brush`, WBTuner has a very close but lower score at the end, although the two have very comparable performance all the time. Fig. 13 shows that the WBTuner's result is not inferior.

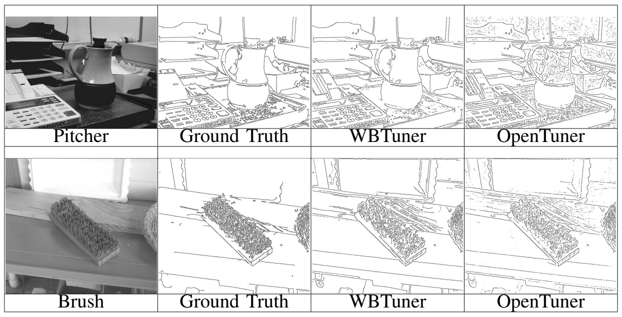

Fig. 13. `Canny` results of WBTuner and OpenTuner

2) *Bioinformatics:*

Phylip. Phylip [31, 58] generates the phylogenetic tree of given protein or DNA sequences by calculating the distances. It shows the evolutionary relationships between various biological species. `Phylip` consists of five stages of computation as shown in Fig. 14.

Stage 1 generates the transition probability matrix and has a tunable parameter `ease`. Stage 2 loads data and performs preprocessing. Stage 3 generates the distance matrix based on the transition probability matrix and the input. It has two tunable parameters `invarfrac` and `cvi`. Stage 4 initializes the phylogenetic tree. Stage 5 generates the tree based on the distance matrix from stage 3. It has a tunable parameter `power`. WBTuner tunes stages 1, 3 and 5. The $wbt\_aggregation()$ primitive is called at the end of stages 1 and 3 with the duplicate-elimination (DEDUP) strategy to prune the sample runs that have similar matrices. Thus, new tuning processes are only spawned for unique matrices. At the end of stage 5, the aggregation selects the tree with the lowest sum of squares, which is the default scoring function. Lower score means the better result.

Fig. 15 shows tuning score comparison for ten datasets from [52] when WBTuner converges. Observe that tuning is critical for this program. On average, WBTuner can reduce the errors by a factor of 283 when compared with no tuning, and by a factor of 4.77 when compared with OpenTuner.

Fig. 16 shows the tuning score variations over time for `data2` and `data10` that have the maximum and minimum improvement over OpenTuner, respectively. For `data2`, 40 seconds of tuning in WBTuner achieves a similar result as 135 seconds of tuning in OpenTuner. The improvement is achieved by the independent tuning/pruning in the three tuning regions. Although OpenTuner outperforms WBTuner for `data10`, the difference between the two results is nearly invisible.

```
void PhyloTreeGeneration() {
  /* STAGE ONE */
  wbt_sampling(numberOfSamples, random);
  ease = wbt_sample(uniform(0.1,0.9))
  maketrans();
  qregigen(prob);
  wbt_expose(probMatrixSize);
  wbt_aggregate(probMatrix, DEDUP);

  /* STAGE TWO: read in data and preprocessing */
  /* STAGE THREE */
  wbt_sampling(numberOfSamples, random);
  cvi = wbt_sample(uniform(0.5,5));
  invarfrac = wbt_sample(uniform(0.1,0.9));
  makedists();
  wbt_expose(distMatrixSize);
  wbt_aggregate(distMatrix, DEDUP)

  /* STAGE FOUR: phylogenetic tree initialization*/
  /* STAGE FIVE */
  wbt_sampling(numberOfSamples, random);
  power = wbt_sample(uniform(1.5,2.5));
  maketree();
  wbt_aggregate(phyTree, MIN, scoring);
  output(phyTree);
}
```

Fig. 14. White-box tuning for `phylip tree`

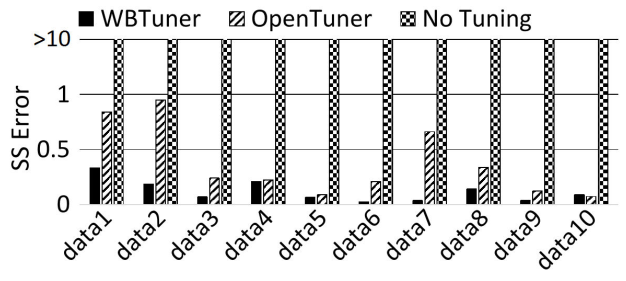

Fig. 15. `Phylip tree` tuning scores on 10 datasets.

3) Machine Learning:

Support Vector Machine (SVM). SVM [25] is a popular machine learning algorithm for data classification. It is a supervised learning technique which takes the training data with feature class labels to build a model for classifying new data. We use the multi-class SVM [36] to classify data with multiple class labels. The algorithm has 8 tunable parameters, which lead to substantially different models if tuned differently. Furthermore, like most machine learning algorithms, certain parameter settings may lead to overfitting (Section IV-A). Thus, we leverage the k-fold cross-validation in WBTuner to tune the parameters while preventing overfitting.

We compare the results tuned by WBTuner with and without cross-validation for 10 datasets obtained from [11]. We divide each dataset into two equal sets and use the first half for training and tuning and the second half for testing. We then collect the results after both tuning converge. The results are depicted in Fig. 17. Observe that for the left two bars (without cross-validation), the training error (black bar) is close to zero while the testing error is very high, indicating overfitting. For the right two bars (with cross-validation), the testing error is significantly lower than that without cross-validation, which strongly suggests that cross-validation substantially mitigates the overfitting problem. That is, the new model generalizes better from the training dataset, without being affected by its details and noise. The results strongly suggest that overfitting is a prominent challenge in tuning and WBTuner effectively addresses this problem transparently.

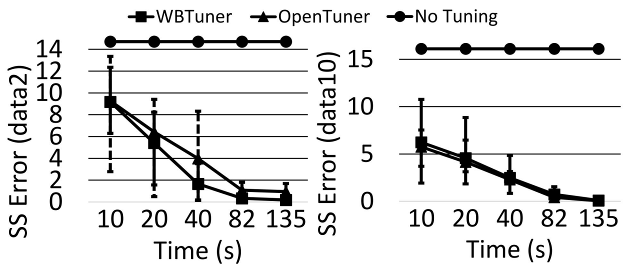

Fig. 16. `Phylip tree` tuning score variation

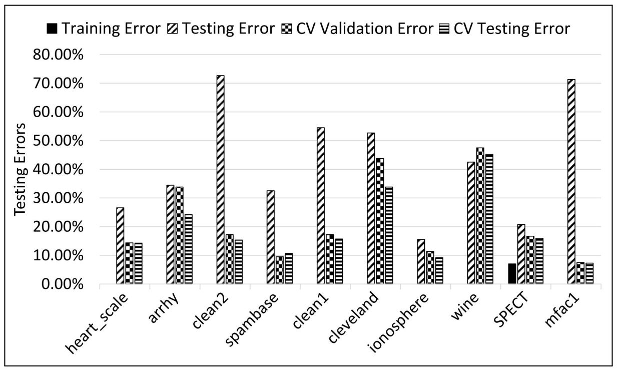

Fig. 17. SVM tuning scores of 10 datasets w/wo validation

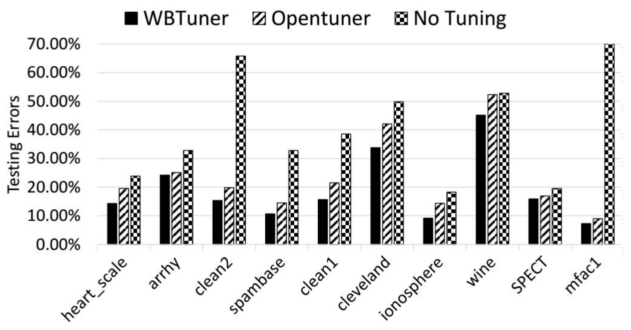

Fig. 18. `SVM` tuning scores of 10 datasets

We also compare the result generated by WBTuner and OpenTuner. As OpenTuner does not handle overfitting by default, we extended its implementation to provide cross-validation as well (using the same k). Observe that WBTuner consistently outperforms OpenTuner. The tuning improvement by OpenTuner over no-tuning is 35% whereas the improvement by WBTuner is 47%. Fig. 19 shows the score variation for the best and the worst datasets. Observe that for Cleveland, even after 1500 seconds, OpenTuner cannot reach the result produced by WBTuner within 80 seconds.

4) Speech Recognition:

Sphinx. Sphinx [43] is a popular speech recognition system. It takes a raw audio and a dictionary, and generates the script for the audio according to the dictionary. It has 16 tunable parameters, such as the upper and lower edges of filers,

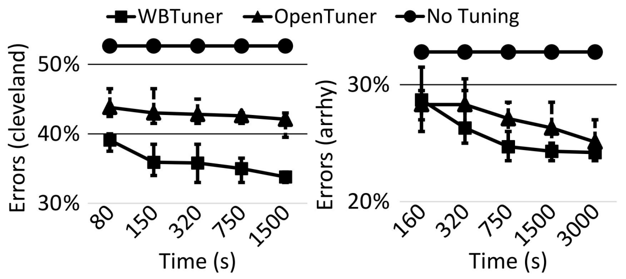

Fig. 19. `SVM` tuning score variation

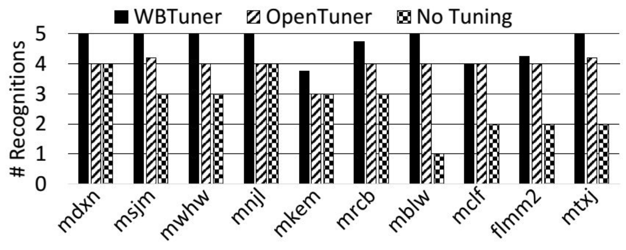

Fig. 20. `Sphinx` tuning of 10 datasets

language weight, and word insertion penalty. These parameters are critical to the recognition results. Different persons' audios may require different parameter sets. Since there does not exist a general scoring function, the tuning results are aggregated using majority vote. OpenTuner is also extended with the majority voting capability for fair comparison.

In the study, we took 10 sets of audios (for 10 persons) from the AN4 dataset [61], each set having 5 audios. We applied both WBTuner and OpenTuner to all these 50 audios. Fig. 20 shows the recognition precision comparison when WBTuner converges (i.e., the number of audios that are correctly recognized for each dataset). Observe that WBTuner precisely recognizes all 5 audios for 6 out of 10 sets, and more than 4 audios for another 3 sets. To reduce non-determinism, we ran the experiment multiple times and took the average. Thus, there are some decimal numbers in the precision results. In contrast, `Sphinx` can only recognize 2.7 audios on average without tuning, and 3.94 audios with OpenTuner. Fig. 21 shows the score variations for the best and worst data sets.

5) Tuning Drone's Behavior.:

Here we demonstrate how we can leverage WBTuner to tune large and complex cyber-physical systems for behavior

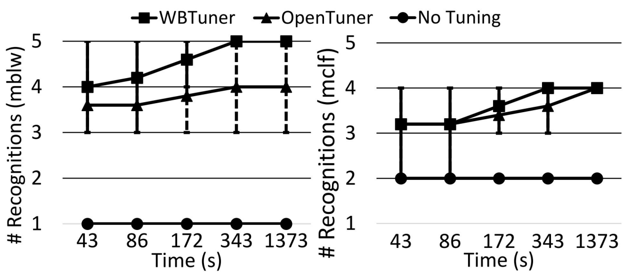

Fig. 21. `Sphinx` tuning score variation

learning. Specifically, we aim to tune one drone's parameters so that it mimics the behavior of the other one.

We use two pieces of widely used drone control software: PX4 [47] and Ardupilot [45]. They are complex (385k and 278k LOC respectively), and have completely different features and implementations. Furthermore, PX4 has 426 configurable parameters and Ardupilot has 612 and the meanings of these parameters are quite different. Thus, High-end drones usually have parameter configurations enabling much better performance, as their engineers spent a lot of time in tuning. For example, Ardupilot flies much slower than PX4 (25% slower) and has much higher battery consumption. PX4s controller is clearly out-performing Ardupilot. WBTuner allows Ardupilot to automatically learn from PX4, saving the substantial manual tuning efforts. Note that there is hardly any correspondence between parameters across the two systems so that one cannot simply copy parameter values. Moreover, only increasing the speed is suboptimal because there are other parameters to consider such as power consumption or waypoint radius to prevent overshoot.

Although both PX4 and Ardupilot provide their own specific black-box parameter tuning tools [68, 69], only a limited number of parameters can be tuned by these tools and thus cannot lead to optimal results. Furthermore, they cannot be applied to achieve more sophisticated tuning tasks such as behavior learning, which is a popular tendency for training autonomous vehicles with different purposes [5, 51, 75].

Tuning Target. We aim to tune the parameters of Ardupilot to make it learn the flying behavior of PX4. We identify 40 parameters that are most relevant to drone control in Ardupilot and mark them as the *tuning variables*. We use the motor speed variables as the *sample result variables* since the drone's behavior is mostly determined by the speed of its four motors. We fly both Ardupilot and PX4 under the same mission, and then employ WBTuner to tune the *tuning variables* in Ardupilot while learning from PX4's flying behavior. Namely, we define the scoring function as the root-mean-square errors of the four motors speed between the two controllers. Furthermore, as a typical mission in Ardupilot often needs to execute under multiple flight modes (e.g., takeoff or land), we define the tuning regions as the individual mode control functions.

To tune Ardupilot according to PX4's behavior, we first fly both Ardupilot and PX4 under 2 different missions. The first one consists of taking off, rising to 10 meters, and finally landing. The second mission makes the drone fly along a 45m route with 3 way points. Our experiments are conducted using the Gazebo simulator [41]. The first mission uses 2000 sample runs, while the second uses 6000 runs given its complexity, each taking 20-30 seconds. Overall, the tuning time is about 42 hours due to real-time simulation. Then we test the subsequent performance of Ardupilot with the tuned parameters under a complex mission, where the drone zigzags and returns to the starting point with a flight distance of 165m.

Detailed training results are provided in the supplementary material [71] Section 2.1.2. Fig. 22 shows the results of the testing mission. After tuning, the motors speed of Ardupilot is quite similar to PX4. Even more, its flight time is reduced from the original 105 seconds to 82 seconds (i.e., 22% fewer). The recorded videos for the test mission are available at [1, 2, 59].

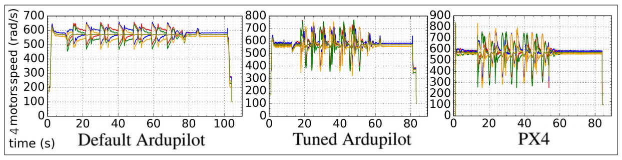

Fig. 22. Testing mission

OpenTuner cannot be applied here for the following reasons. (1) Several parameters that affect multiple flight modes in a single mission. They are tuned to different values for various modes. This cannot be supported by blackbox tuning; (2) Each sample run in OpenTuner is a whole execution that includes expensive simulator startup and drone preparation taking 3-4 minutes per sample. In contrast, WBTuner tunes small code regions and each sample run is just 20-30 seconds; (3) The simulator often fails to start (we suspect that it results from the locked resources of previous closed execution). This is not a problem for WBTuner as it can spawn all the sampling/tuning processes after a successful start.

VI. Related Work

WBTuner is related to many existing input selection or fuzzing works [21, 22, 26, 27, 32, 44, 49, 56]. They use different search techniques such as MCMC or genetic algorithms to address software engineering or cyber-security problems.

Several autotuning frameworks are proposed for domain-specific programs. For example, [14, 19] tune data-mining algorithms; [72] aimed to generate an optimized matrix multiply routine by empirical autotuning; [24] is specialized for tuning stencil computation; and [35] is a stochastic approach for parameter tuning of SVM. In [10], a compiler autotuning framework is proposed to speed up application performance using bayesian networks. Several dynamic autotuning frameworks [3, 8, 13, 15, 23, 34, 62, 63] were proposed to monitor program execution to guide the program to perform self-adaptation for achieving specific optimization goal. For example, PowerDial [34] transforms static configuration parameters into dynamic controllable variables to make programs power-aware.

PetaBricks [6, 9] proposes a language- and compiler-based solution for tunable algorithm construction. Different algorithms and parameter configurations are being tuned to achieve better performance and accuracy. Different algorithms are selected for execution by the Petabricks runtime. It advocates the concept of tuning by construction, targeting on stream data processing. The individual streaming components only interact through their interfaces and do not have any other inter-dependences. However, it cannot tune pre-existing non-streaming programs where inter-dependences across phases are substantial like in Ardupilot. Furthermore, users need to use the proposed language.

Automatic parallelization [4, 66, 67] transforms a sequential program to its concurrent version which is guided by annotations. They divide a computation task into concurrent sub-tasks. In contrast, WBTuner spawns processes to compute similar but different tasks.

VII. Conclusion

We propose WBTuner, a general white-box tuning engine. It provides primitives that allow users to easily compose complex tuning tasks as if they are writing extensions to the original programs. Our experiments show that WBTuner substantially improves data processing results and outperforms the state-of-the-art black-box tuning engine.

Acknowledgements

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1748764 and 1409668, ONR under contracts N000141410468 and N000141712947, and Sandia National Lab under award 1701331. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] Adrupilot-default. ”https://drive.google.com/open?id=0BxgPTM7nEUyCcTFKdjM4RVNrMk0”, 2018.

[2] Adrupilot-tuned. ”https://drive.google.com/open?id=0BxgPTM7nEUyCYmVPdzBtMnoyT1U”, 2018.

[3] Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical report, MIT, 2009.

[4] Jonathan Aldrich, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, and Roger Wolff. Permission-based programming languages (nier track). In *ICSE 2011*.

[5] Olov Andersson, Mariusz Wzorek, and Patrick Doherty. Deep learning quadcopter control via risk-aware active learning. In *AAAI 2017*.

[6] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *PLDI 2009*.

[7] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Una-May OReilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *PACT 2014*.

[8] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O’Reilly, and Saman Amarasinghe. Siblingrivalry: Online autotuning through local competitions. In *CASES 2012*.

[9] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *CGO 2011*.

[10] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization*, 2016.

[11] Arthur Asuncion and David Newman. Uci machine learning repository, 2007.

[12] Exponential backoff. IEEE Standard 802.3-2008, 2008.

[13] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. *SIGPLAN Not.*, 2010.

[14] Jérémy Besson, Christophe Rigotti, Ieva Mitasiunaite, and Jean-François Boulicaut. Parameter tuning for differential mining of string patterns. In *ICDMW 2008*.

[15] V. Bhat, M. Parashar, Hua Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed. Enabling self-managing applications using model-based online control strategies. In *ICAC 2006*.

[16] T. Blaschke. Object based image analysis for remote sensing. *ISPRS Journal of Photogrammetry and Remote Sensing*, 2010.

[17] Dan Bloomberg. Leptonica image processing and analysis library. "http://www.leptonica.com/", 2001.

[18] David S Bolme, J Ross Beveridge, Marcio Teixeira, and Bruce A Draper. The csu face identification evaluation system: its purpose, features, and structure. In *Computer Vision Systems*. 2003.

[19] Ole Burmeister, Markus Reischl, Georg Bretthauer, and Ralf Mikut. Data mining analyses with the matlab toolbox gait-cad. *Automatisierungstechnik*, 2008.

[20] J Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986.

[21] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA 2013*.

[22] Michael Carbin and Martin C. Rinard. Automatically identifying critical input regions and code in applications. In *ISSTA 2010*.

[23] Fangzhe Chang and Vijay Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Cluster Computing, 2011*, 2011.

[24] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS 2011*.

[25] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 1995.

[26] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *ISSTA 2010*.

[27] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *PLDI 2015*.

[28] Martin Ester, Hans Kriegel, Jorg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD 1996*.

[29] A. M. Fahim, A. M. Salem, F. A. Torkey, and M. A." Ramadan. An efficient enhanced k-means clustering algorithm. *Journal of Zhejiang University SCIENCE A*, 2006.

[30] Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michèle Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence*, 2010.

[31] Nir Friedman, Matan Ninio, Itsik Pe'er, and Tal Pupko. A structural em algorithm for phylogenetic inference. *Journal of Computational Biology*, 2002.

[32] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *ICSE 2009*.

[33] Michael D. Heath, Sudeep Sarkar, Thomas Sanocki, and Kevin W. Bowyer. Robust visual method for assessing the relative performance of edge-detection algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1997.

[34] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. *ACM SIGPLAN Notices*, 2011.

[35] F. Imbault and K. Lebart. A stochastic optimization approach for parameter tuning of support vector machines. In *ICPR 2004*.

[36] T. Joachims. Making large-scale SVM learning practical. *Advances in Kernel Methods - Support Vector Learning*, 1999.

[37] Frank Kane. Hands-on data science and python machine learning, 2017.

[38] George Karypis and Vipin Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[39] F. Kerouh. A no-reference blur image quality measure based on wavelet transform. *IJDIWC 2012*.

[40] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 1952.

[41] Nate Koenig and Andrew Howard. Gazebo. "http://gazebosim.org/", 2009.

[42] Paul Lamere, Philip Kwok, Evandro Gouvea, Bhiksha Raj, Rita Singh, William Walker, Manfred Warmuth, and Peter Wolf. The CMU sphinx-4 speech recognition system. In *ICASSP 2003*.

[43] Man-Lap Li, Ruchira Sasanka, Sarita V Adve, Yen-Kuang Chen, and Eric Debes. The alpbench benchmark suite for complex multimedia applications. In *IISWC 2016*.

[44] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. In *ICSE 2012*.

[45] Randy Mackay. Ardupilot. "http://ardupilot.org/", 2007.

[46] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of*

the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics, 1967.

[47] Lorenz Meier. Px4 autopilot. ”http://px4.io/”, 2009.

[48] Peter Merz and Bernd Freisleben. A genetic local search approach to the quadratic assignment problem. In *ICGA 1997*.

[49] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 1990.

[50] Andrew Moore. Cross-validation for detecting and preventing overfitting, 2001.

[51] Igor Mordatch, Kendall Lowrey, Galen Andrew, Zoran Popovic, and Emanuel V. Todorov. Interactive control of diverse complex characters with neural networks. *Advances in Neural Information Processing Systems*, 2015.

[52] Ramanathan Narayanan, Berkin Özisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. Minebench: A benchmark suite for data mining workloads. In *IISWC 2006*.

[53] K A Abdul Nazeer, S D Madhu Kumar, and M P Sebastian. Enhancing the k-means clustering algorithm by using a o(n logn) heuristic method for finding better initial centroids. In *EAIT 2011*.

[54] K A Abdul Nazeer and M P Sebastian. Improving the accuracy and efficiency of the k-means clustering algorithm. In *WCE 2009*.

[55] Xia Ning and George Karypis. Slim: Sparse linear methods for top-n recommender systems. In *ICDM 2011*, 2011.

[56] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE 2007*.

[57] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *PNAS 1998*.

[58] DOTREE Plotree and DOTGRAM Plotgram. Phylip-phylogeny inference package (version 3.2). *Cladistics*, 1989.

[59] PX4. ”https://drive.google.com/open?id=0BxgPTM7nEUyCYnRxS2FSN2JRbEE”, 2018.

[60] J Ross Quinlan. *C4. 5: programs for machine learning*.

[65] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society. Series B (Methodological)*, 1974.

1993.

[61] Raj Reddy. An4 database. ”http://www.speech.cs.cmu.edu/databases/an4/”, 1991.

[62] Michael F Ringenburg, Adrian Sampson, Luis Ceze, and Dan Grossman. Profiling and autotuning for energy-aware approximate programming. In *WACAS 2014*.

[63] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 2009.

[64] James C. Spall. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Transactions on Automatic Control*, 1992.

[66] Kevin Streit, Johannes Doerfert, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. Generalized task parallelism. *TACO*, 2015.

[67] Dinda Findler Swaine, Tew and Matthew Flatt. Back to the futures: incremental parallelization of existing sequential runtime systems. In *ACM Sigplan Notices*, 2010.

[68] Ardupilot Tuning. ”http://ardupilot.org/copter/docs/tuning.html”, 2017.

[69] PX4 Tuning. ”https://docs.px4.io/en/advanced_config/pid_tuning_guide_multicopter.html”, 2017.

[70] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 2004.

[71] WBTuner. Wbtuner source and supplementary material. ”https://github.com/cgo2019/WBTuner”, 2018.

[72] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *SC 1998*.

[73] Madhu Yedla, Srinivasa Pathakota, and T M Srinivasa. Enhancing k-means clustering algorithm with improved initial center. *IJCIT 2010*.

[74] Fang Yuan, Zeng-Hui Meng, Hong-Xia Zhangz, and Chun-Ru Dong. A new algorithm to get the initial centroids. In *Proceedings of the 3rd International Conference on Machine Learning and Cybernetics*, 2004.

[75] T. Zhang, G. Kahn, S. Levine, and P. Abbeel. Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. In *ICRA 2016*.