White-Box Tuning for Better Data Processing

Abstract

Data processing programs are largely parameterized, especially those based on heuristics. The quality of the generated outputs depends on the parameter setting. Different inputs often have different optimal settings. Parameter tuning is hence of great importance. Existing tuning techniques treat the data processing 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 tuning tasks by adding a small number of library calls to the original program and providing a few callback functions. Our experiments on 14 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.

1. Introduction

Data processing programs are becoming increasingly important in the Big-data era. Their complexity is also growing at an enormous pace, involving more and more computation stages. A prominent challenge is that many of them are parameterized, meaning that the user has to configure a set of parameters before running these programs. More importantly, the optimal configuration is mostly dependent on the specific input. Different inputs require different configurations in order to achieve the best results.

For instance, the results of *k-means* [38], a well-known data clustering algorithm, heavily depends upon the choice of k that is the number of clusters into which the user wants to partition the input data. A lot of research [22, 43, 44, 59, 60] has aimed at automatically deriving the appropriate k value from the input. However, as far as we know, there is no general solution for finding k. Another example is object detection in satellite image processing [15]. It is a computation-intensive and parameterized procedure that has to deal with a large volume of images in a time unit. 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` [17], one of the most widely used image processing algorithms that detect edges. It is a multi-staged algorithm with three important parameters. It is well-known that `Canny`'s results heavily depend on the provided parameters. According to [27], 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`. Left two are original images. Others 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 two (1.8, 0.2, 0.7) produces the better result for the trashcan. Therefore, automated parameter tuning becomes critical in data processing as manual tuning is not realistic.

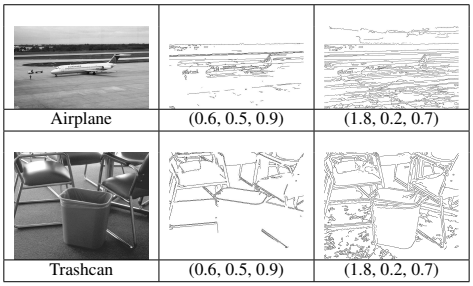


Figure 1: `Canny`**'s results with different parameters**

Key Observation of Staged Computation Paradigm By observing `Canny` and many other real world data processing applications, we find that they typically follow the *staged computing paradigm*, i.e., they consist of multiple computation stages such that each stage have a unique set of tunable parameters.

Existing Work Multiple frameworks have been proposed to automate the 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 the user-provided scoring function of the final result, they sample the parameter space to find the best parameter configuration. Internally, they may adopt the Stochastic algorithms [33, 34, 51, 52] or Genetic algorithm [40] as the search strategies. While the above frameworks have been proven to be effective, they suffer greatly from poor performance due to the *inherent* limitations of the *black-box* designs:

- *All* parameters are included in each parameter configuration, which leads to the *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 large corpus of data and conduct the preprocessing, which are very time consuming.

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, it spawns multiple processes to sample different parameter configurations involved by a stage. At the end of each stage, it aggregates the sampled internal results of that stage through a default or custom (provided by the user) aggregation strategy. The aggregation step reduces the processes to one or a small number of processes (with good internal results), which will proceed to tune the next stage. Intuitively, the aggregation strategy may either select the min/max value from all internal results or merge them as the average value (Sec. 4.3).

Consider an application with n stages of computation with each stage having one unique parameter to tune, whose domain has m unique values. Initially, WBTuner spawns m sampling processes to cover m configurations of the first stage. Assum-

ing the ideal case that only one of the processes from one stage proceeds to the next stage, 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 alive 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.

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.
- The aggregation lets us discard early the redundant (or bad) computations that follow the certain internal results, which would be performed by the black-box approaches.
- 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. Remember that every full execution needs to load and pre-process large corpus of data, which only have to be done once in WBTuner.

TODO: we will work on contritutions finally. Our key contributions include the following.

- We propose white-box tuning that treats the subject program as a white-box and allows the user to access and manipulate the internal program states during tuning. It is complementary to black-box tuning, especially when the user has access to the source code and a certain level of domain knowledge.
- We develop a prototype WBTuner in the form of library. The library provides a set of expressive tuning primitives and an advanced runtime system to compose complex execution models. The runtime hides most of the underlying complexity such as process management, data transfer, and output aggregation from the user, who can hence focus on writing the high level tuning logic.
- We use WBTuner to tune 13 widely used data processing programs for image processing, data mining, machine learning, pattern matching, and bioinformatics. Our experiments show that WBTuner can substantially improve the data processing results with reasonable overhead. The comparison with OpenTuner shows that in some cases, OpenTuner can never achieve the same tuning results (scores differences $> 10\%$) as WBTuner. In the other cases, OpenTuner takes 3.08X time to achieve the same results under a single core environment and 4.35X when multiple cores are used.
- We use WBTuner to tune the parameters of a large drone controller software (278K LOC) to mimic the behavior a different controller with a better algorithm.
- We release our implementation of WBTuner for the community at [57].

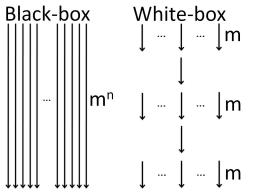


Figure 2: Execution models of black-box and white-box tuning

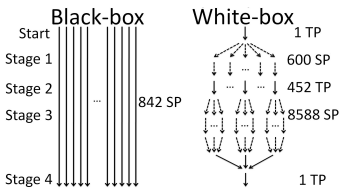


Figure 3: Tuning `Canny`. TP/SP are tuning/sampling processes.

2. Overview of White-Box Tuning Framework

We present an overview of WBTuner using `Canny`, a popular image processing algorithm as shown in Fig. 5.

Running Example It has four stages: the *Gaussian smoothing* stage (line 22 in Fig. 5) which removes the noise from the image, the *image transformation* stage (line 30) which performs non-maximal suppression, the *edge traversal* stage (line 37) which leverages hysteresis analysis to track all potential edges in the image, and the *visualization* stage which visualizes the final results.

`Canny` 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 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 have a unique set of tunable parameters.

User Interface WBTuner provides the users with an intuitive interface, which consists of multiple tuning primitives (i.e., library calls), as shown in Fig. 4. Note they are implemented with the same programming language as the original program, rather than some additional specification language.

Library Calls :	**@sampling**$(n, cbStrgy)$ | **@aggregate**$(x, cbAggr)$ | **@sample**$(x, cbDist)$ | **@expose**(x) | **@load**(x) | **@loadS**(x, i) | **@split**$()$ | **@sync**$(cbBarrier)$ | **@check**$(cbChk)$
CallBack :	*cbStrgy, cbAggr, cbDist, cbChk, cbBarrier*

Figure 4: Primitives

Fig. 5 shows how the interface is used (Note the 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 sample input variable

(i.e., a variable to tune). It also specifies the distribution of the variable from which sample values are taken.

Function `AggregateGaussian()` is a callback function provided by the user to facilitate tuning. In this example, we implement it following an existing approach [32] to prune the poorly smoothed ones. Specifically, it loads (line 6) the images denoted by `sImage` which are sampled in different parameter settings of `sigma` and determines (line 7) whether each image is properly smoothed given the image size `imgSize`. We will explain the relevant primitives `wbt_loadS`, `wbt_loadS` and `wbt_expose` in Section 3.3. 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 `sigma` value used to produce the image. Next we will discuss the runtime 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(19, 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 }
```

Figure 5: White-box tuning for `Canny`. The highlighted statements are added. Tuning primitives start with `wbt`.

Runtime Execution Model The runtime execution framework is shown in Fig. 6. 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 the results 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. 3.2). When a sampling process encounters a tuning variable, 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 submitting driver* (⑤) to commit its own computation results and terminates. After all sampling processes commit, the tuning process resumes and invokes the *the parent aggregation driver* to aggregate the sampling results (⑥). It then continues to execute normally with the aggregated results (⑦).

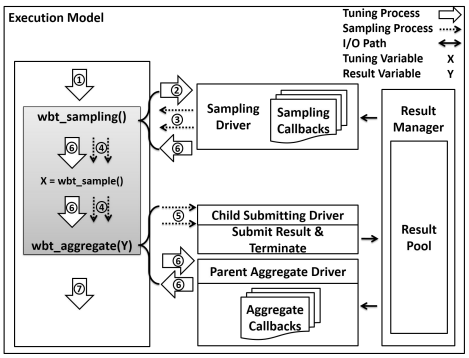


Figure 6: Execution Model

The above simplified model assumes a single tuning process in the runtime system. It is usually necessary to have multiple tuning processes. For example, consider the aggregation at line 27 in Fig. 5, 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 achieve this, the user can use our 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 support multiple tuning processes (Section 3.2).

Result and Comparison Initially we have 600 samples (line 20). At the end of the Gaussian smoothing stage (line 27), the invoked function `AggregateGaussian()` prunes 148 samples that are not properly smoothed, therefore only 452 samples need to be kept. 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 19 samples [1] (with different configurations of the parameters `low` and `high`) for each smoothed image. Hence the total number of samples is $452 \times 19 = 8588$.

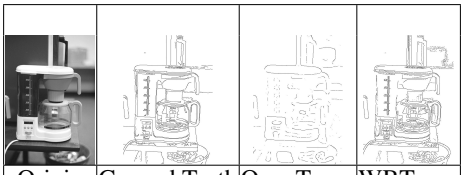
The sampling results are aggregated by majority voting (line 41), that is, a pixel is set if it is set in more than 50% of the

[1] The reason we use a smaller number of samples in this stage is that Canny is less sensitive to these thresholds.

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`. 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 default search strategy in OpenTuner which is called the Multi-armed bandit search. 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 8588 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 [56] to compare the result with the ground truth result hand-picked by experts [27]. Both visual and scoring results demonstrate that that WBTuner outperforms OpenTuner.



Origin	Ground Truth	OpenTuner	WBTuner
samples		842	8588
SSIM	1	0.592	0.794

Figure 7: Tuning `Canny` with image coffeemaker in 90s.

3. 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 *processes* and *stores*. 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. Furthermore, each sampling process produces a lot of sample data from internal states. Storing, accessing, and aggregating such data (across processes) is also challenging. All these complexities should be transparent to the users. In this section, we present our runtime system along with the formal semantics.

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

3.1. Stores

WBTuner has two stores, the *store* σ for regular program states and the *sample store* δ that is shared across all processes to store sampling outputs. The two are isolated. Any state transfers between the two are performed explicitly. In δ, states can be divided into two classes: (1) exposed store, a store for exposed variables, (2) aggregation store, a store for sampled results from the children processes.

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 even outside its local scope (e.g., function). Therefore, the exposed local variable can be used to pass the value across different scopes. For instance, in Fig. 5, the local variable `imgSize` from the `canny` function is exposed at line 26 and then loaded at line 7 in the `AggregateGaussian` function.

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.

Aggregation Store Aggregation store of a tuning process stores the sampled values from the children processes. 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. **Check if this sentence is correct. From regular store or some other store?** At the semantic level, the primitive *wbt_aggregate*$(x,\dots)$ forces each child process to write the value of x in its regular store to the aggregation store of the tuning process, as illustrated by line 27 in Fig. 5. The primitive *wbt_loadS*(x,i) loads the value of x from the ith child process, as illustrated by line 6 in Fig. 5.

Our system achieves the semantics by leveraging the file system in disk. In particular, a sampled value is stored as a file, of which the name is in the form *var_pid*, where *var* specifies the name of the variable (that holds the sample result) and *pid* specifies which child process submits the value of the variable. All the files are stored in a directory owned by the tuning process. To load data 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).

3.2. Processes

WBTuner supports two execution modes, $\mathbb{T}\langle pid\rangle$ denotes the current process *pid* is a tuning process whereas $\mathbb{S}\langle pid\rangle$ a sampling process. To facilitate discussion, we also extend the statements to include a **spawn**(σ,δ,ω,s) statement that forks a process with the specified stores, execution mode, and the

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

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:

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

Figure 8: Operational Semantics

process body s, a **notify**(pid) statement that notifies a process pid, a **wait**(pid) statement that wait for a notification from the process pid, and an **invoke**(**cb**) statement that invokes a callback function cb.

Process Scheulding In practicem 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 without sacrificing the tuning performance greatly. 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. **What is watermark? is this a standard term ???** At line 8, a threshold is computed to denote a watermark of resources. If the available resources are below the watermark, then 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 process. 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.

3.3. Primitives

Rule [*SAMPLING*] forks n sampling processes (indicated by the $\mathbb{S}\langle i\rangle$ mode) through the **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, "**invoke**$(cbStrgy);s$". After forking, callback $cbStrgy()$ is called to initialize the sampling strategy in the children. We want to point out 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 runs 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. Then the sampling process 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 **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. 5.2.3), which improves not only the performance but also the final clustering results. Note thats such improvements are impossible to achieve in black-box tuning.

Rule [*EXPOSE*] exposes the values 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 sample store) inside some callback function in a tuning process. Rule [*LOADSAMPLE*] loads the sample outcome of x from the ith sample run.

4. Practical Challenges

Algorithm 1 Process Scheduling

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

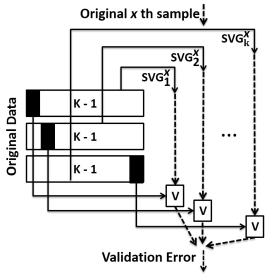


Figure 9: Tuning-validation Model for the xth sample run

4.1. Overfitting

Since machine learning algorithms normally produce models as their output, the tuning task of these parameters is usually guided by the execution results of the models (e.g., lower classification errors). Unfortunately, it may lead to *overfitting*, meaning that the tuned parameters produce optimal results on the training data but poor results on the testing data. Note that other programs (e.g., `Canny`) do not have this problem as they are tuning for the final output data but not models tested by different data.

WBTuner provides intrinsic support to address overfitting by combining its execution model with *k-fold cross-validation* [53], a widely used technique for preventing overfitting. 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. By doing so, WBTuner will then transparently include k-fold cross-validation during tuning.

The tuning-validation model is shown in Fig. 9. First, the input data is transparently divided to k datasets. For each of the 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 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 result in Fig. 21 demonstrates the necessity for cross-validation.

4.2. 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, it entails massive storage and I/O overhead. We observe that many benchmarks aggregation can be performed incrementally as they involve functions such as finding the min, max, average, or majority (voting). For instance, for the aggregation strategy min, each sampling process updates a shared global min by comparing its outcome to it. To support incremental averaging, WBTuner uses a shared ring buffer to which sampling processes copy their results. The tuning process consumes the data from the buffer to perform incremental averaging. Majority voting is handled in a similar fashion. Fig. 10 demonstrates that incremental aggregation substantially reduces the tuning time and memory consumption for WBTuner.

4.3. Sampling/Aggregation Strategies

In addition to custom strategies provided by the user, WBTuner supports several common sampling/aggregation strategies. The user only have to denote the name of the strategy 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, there are min, max, majority vote (MV), averaging (AVG), and duplicate elimination (DEDUP). These strategies are normally enough for most of the tuning task according to our experience. Observe that only four benchmarks use custom aggregation strategies out of 14 benchmarks.

4.4. Auto-tuning Sampling Number

Because the number of samples varies from one tuning region to another, WBTuner provides an automatic way similar to exponential backoff [11] to determine the number of required samples. For the provided sampling number in each primitive *wbt_sampling*(), WBTuner first checks whether the result is better by doubling it. If yes, then the number of samples is doubled again until the sampling result converges.

4.5. Specification

The specification offers the users great flexibility in designing various tuning strategies by granting them the access to the internal program states. Meanwhile, we argue the specification overhead is modest. First, based on our experience, it is easy to specify the top-level stages [2], which suffice to speed up the performance measurably. Second, we summarize the common patterns of the aggregation strategies (Sec. 4.3), which cover the majority of the use cases.

5. Evaluation

We evaluate the efficiency and effectiveness of our WBTuner implementation in C and compare it with OpenTuner. Experiments were run on a machine with Intel i7-2640M 2.80GHz processor and 16GB RAM.

Benchmarks.

A wide variety of benchmarks are used in our experiment, including 13 widely used data processing programs and an open-source controller software for commercial drones. These are heavily parameterized applications. For further benchmarks information, please refer to the technical report [57].

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 1. Most benchmarks come with their own scoring functions, so the callbacks for them are implemented accordingly. Results comparison of benchmarks without scoring functions (i.e., with superscript 1 in Tab. 1) is explained in section 5.1.

Column 1-2 show programs names and 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 using a different sampling strategy (MCMC). `C4.5` and `SVM` use random sampling together with cross-validation. Cross validation is also implemented in OpenTuner for these two benchmarks. Column 7 presents the lines of code 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 5.2. The LOCs for callbacks are small compared to the source code LOCs. They mainly implement scoring functions or checks.

5.1. Tuning Results Summary

In the first experiment, we ran each benchmark with the largest dataset under three settings – (1) native run without tuning; (2) black-box tuning using OpenTuner; (3) white-box tuning using WBTuner – and observed the best possible tuning results and the corresponding execution time. Default search strategy, the *multi-armed bandit* [23] is used in OpenTuner.

We ran WBTuner with the number of samples auto-tuned by WBTuner until converging, then we collected the tuning time. For OpenTuner, we gradually increased the timeout parameter until it either reaches the similar results (difference $< 10\%$) as WBTuner or could not reach the similar results after spending 10 times of the tuning time of WBTuner. 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 1). To achieve fair comparison, for these benchmarks, we implemented 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 1, 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 time of tuning execution 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 10 times 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

[2] We plan to combine the program analysis and natural language processing to automatically identify the fine-grained stages.

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
↑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	-
↑Watershed [1]	270k	3	5	RAND	MV	34	1.03	0.41	26.1	0.65	31.5	0.65	2.11	0.93	7.8	0.65	30.81	0.65	3.95
↑Kmeans	468	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
↑DBScan	408	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
↓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
↑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	-
↓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	493.21	0.84	2.33
↑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	-
↑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
↓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
↑L2AP	37.2k	3	5	RAND	CUSTOM	39	0.48	2018	1.21	2170	3.4	2170	2.8	0.362	0.84	2170	2.03	2170	2.42
↓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
↓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
↓Ardupilot	278k	20	22	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 score of WBTuner.

Table 1: Benchmark statistics and the experiment results for achieving the best tuning scores.

out WBTuner in 2 out of the 14 cases. For the other cases, the average tuning overhead of OpenTuner is 3.08X higher than WBTuner. For multi-core environment, 3 programs time out and the overhead ratio is 4.35X.

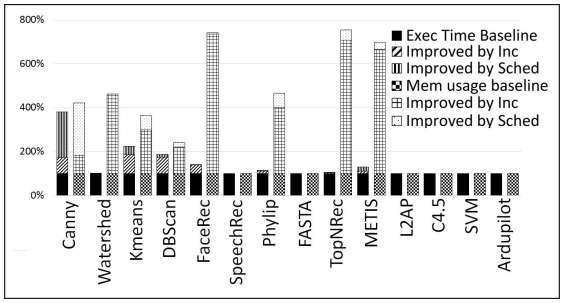


Figure 10: Optimization effects on different benchmarks

Observe that WBTuner substantially improves the result quality compared to without tuning and it is much more effective than OpenTuner. For the cases that OpenTuner can reach the scores of WBTuner, we have also given more tuning time to OpenTuner. But OpenTuner did not produce better tuning results.

Fig. 10 shows the effect of the various optimizations discussed in Sec. **??** and Sec. 4.2. The baseline is the time/memory consumption *after optimizations*. Observe that the incremental aggregation is highly effective for several cases, especially for reducing the memory consumption as it prevents reading large number of results for one-shot aggregation. Observe that the scheduler further improve the performance in several cases, especially for Canny and K-means.

5.2. Tuning Case Studies

In this section, we study the details of tuning several representative programs under the single core environment and tuning Ardupilot in the multi-core environment.

5.2.1. Image Processing

Canny. In Section 2, we have already shown the tunning results of `Canny`. Here we used 10 different images from [27], where each image comes with 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 [56] 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 (see Fig. 5).

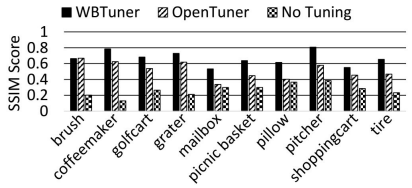


Figure 11: `Canny` tuning scores of 10 images.

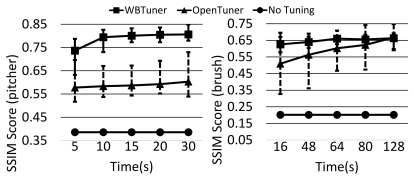


Figure 12: Canny tuning score variation

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. And Fig. 13 seems to indicate the WBTuner's result is not inferior.

5.2.2. Bioinformatics

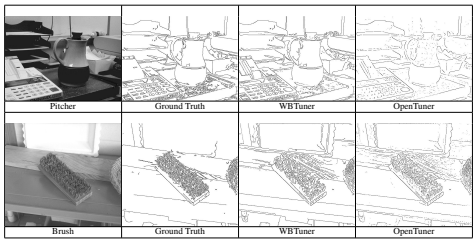


Figure 13: `Canny` tuning results of WBTuner and OpenTuner

Phylip. Phylip [24, 46] generates the phylogenetic tree of given protein or DNA sequences by calculating the distances. It show the evolutional relationships between various biological species. `Phylip` consists of five stages of computation as shown in Fig. 14.

Stage 1 is for transition probability matrix generation. It 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 at the end of stages 1 and 3 are called with duplicate-elimination (DEDUP) strategy to prune the sample runs that have similar matrices. Thus, child tuning processes are only spawned for unique matrices. At the end of stage 5, the aggregation is to select the tree with the lowest sum of squares, which is the default scoring function. Lower score means the better result.

```
1  void PhyloTreeGeneration() {
2    /* STAGE ONE */
3    wbt_sampling(numberOfSamples, random);
4    ease = wbt_sample(uniform(0.1,0.9))
5    maketrans();
6    qregigen(prob);
7    wbt_expose(probMatrixSize);
8    wbt_aggregate(probMatrix, DEDUP);
9
10   /* STAGE TWO: read in data and preprocessing */
11   /* STAGE THREE */
12   wbt_sampling(numberOfSamples, random);
13   cvi = wbt_sample(uniform(0.5,5));
14   invarfrac = wbt_sample(uniform(0.1,0.9));
15   makedists();
16   wbt_expose(distMatrixSize);
17   wbt_aggregate(distMatrix, DEDUP)
18
19   /* STAGE FOUR: phylogenetic tree initialization*/
20   /* STAGE FIVE */
21   wbt_sampling(numberOfSamples, random);
22   power = wbt_sample(uniform(1.5,2.5));
23   maketree();
24   wbt_aggregate(phyTree, MIN, scoring);
25   output(phyTree);
26 }
```

Figure 14: White-box tuning for phylogentic tree generation

Fig. 15 shows tuning score comparison for ten datasets from [42] 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.

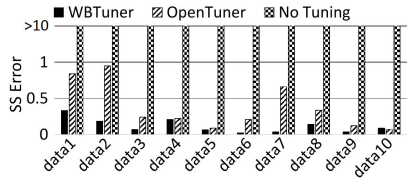


Figure 15: Phylogenetic tree tuning scores on 10 datasets.

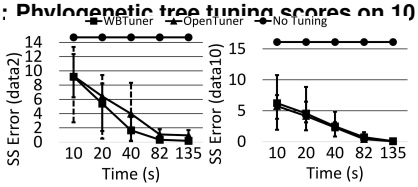


Figure 16: Phylip tuning score variation

5.2.3. Data Mining / Machine Learning

K-means. K-means [38] partitions input dataset into `K` clusters, each holding similar data. The algorithm is shown in Fig. 17. Given the input `objs`, it first generates `K` initial centroids (for clusters) through a non-deterministic random procedure (line 14). It then calls `kmeansCore()` (line 17) to iteratively refine the centroids and the corresponding clusters. The clustering result is stored in `objsAssign`. A data object is put in the cluster of the closest centroid. Although there is only one tunable parameter, `K`, in `K-means`, the clustering result is nondeterministic even `K` is fixed because it also depends on the randomly selected initial centroids.

The tuning code is highlighted in Fig. 17. The quality of clustering result is calculated by the Silhouette score [49]. Higher score means better clustering result. We use MCMC sampling (line 11) to demonstrate the use of a different sampling strategy. It computes the probability of accepting the current sample result by computing its improvement over the old score. If a sample is accepted, the next sample will be in its neighborhood. Lines 2-5 denote the scoring function for one sample. Observe that it is used for both sampling ad aggregation strategy. Note that the scoring function implementation is trivial because Silhouette score calculation is provided by the benchmark. Furthermore, the rest of MCMC sampling and MAX aggregation is transparent. In addition, line 15 invokes a callback to prune the centroids with poor quality based on the algorithm in [44], which prevents the expensive core algorithm execution. Totally there are 5 lines of primitives and 98 lines in callbacks.

```
1  /* User provided callback */
2  double scoring(int sampleIdx) {
3    result = wbt_loadS(objsAssign, sampleIdx)
4    return silouette(result);
5  }
6  /* Source program */
7  void kmeans() {
8    // Initializations
9    objs = input(file);
10
11   wbt_sampling(numberOfSamples, mcmc, scoring)
12   K = wbt_sample(uniform(2, 25))
13   initializeCentroids(centroids);
14   wbt_expose(centroids)
15   wbt_check(checkCentroids)
16
17   kmeansCore(objs, objsAssign, centroids, K);
18   wbt_aggregate(objsAssign, MAX, scoring)
19   visualize(objsAssign);
20 }
```

Figure 17: White-box tuning for `K-means`

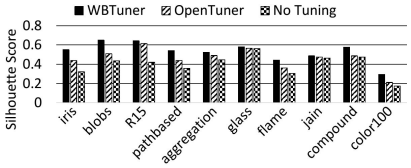


Figure 18: `K-means` tuning scores for 10 datasets

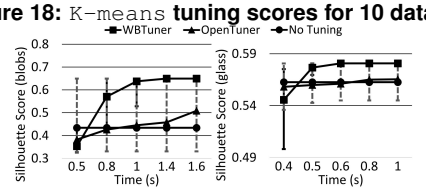


Figure 19: `K-means` tuning score variation

Fig. 18 shows the clustering results when the tuning time is the convergence time of WBTuner. The ten data sets are from [19, 25, 26, 30, 36, 45, 55, 61]. Observe that WBTuner consistently produces the best results. The result improvement with OpenTuner over no tuning is 7% on average. The improvement with WBTuner is 38%. In other words, WBTuner is almost six times more effective in tuning the results with the same time. Fig. 19 shows the score variations for the `blobs` and `glass` datasets, in which WBTuner has the maximum and minimum improvement over OpenTuner. Fig. 20 visualizes the different clustering results. Observe that the WBTuner result is more reasonable. For `glass`, although the two yield very similar final results, WBTuner reaches the result within 0.6 second whereas OpenTuner cannot reach the same score in 1 second.

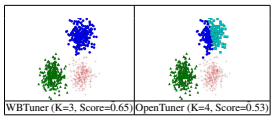


Figure 20: Blobs clustering

Support Vector Machine (SVM). SVM [21] is a widely used machine learning algorithm for data classification and regression analysis. It is a supervised learning technique which takes the training data with feature class labels to build a model, used to classify new data later. We use the multi-class SVM [31] 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 4.1). Thus, we leverage WBTuner's k-fold cross-validation to tune the parameters while preventing overfitting (with k=10, a typical setting for cross-validation [53]).

We compare the results tuned by WBTuner with and without cross-validation for 10 datasets obtained from [10]. 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. 21. Observe that although cross-validation tuning produces much higher training errors than tuning without cross-validation, it has much better testing results and thus 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 problems transparently.

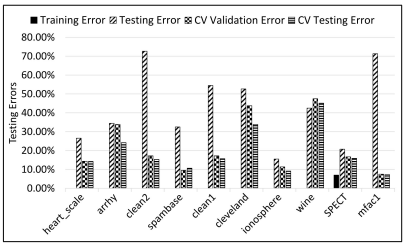


Figure 21: SVM tuning scores of 10 datasets w/wo validation

We also compare the result generated by WBTuner and OpenTuner. As OpenTuner does not handle overfitting by default, we extended its implementation to provide the 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. 23 shows the score variation for the best and worst datasets. Observe that for Cleveland, even after 1500 seconds, OpenTuner cannot reach the result produced by WBTuner within 80 seconds.

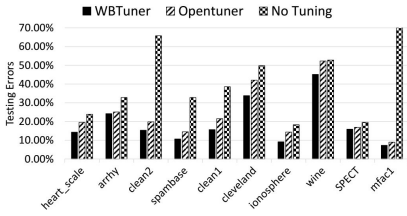


Figure 22: SVM tuning scores of 10 datasets

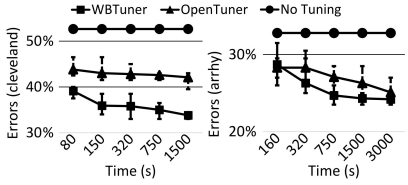


Figure 23: SVM tuning scores variation

5.2.4. Training Drone's Behavior.

In this case study, we demonstrate how we can leverage WBTuner to tune large and complex cyber-physical systems for behavior 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 [39] and Ardupilot [37]. They are complex (385k and 278k LOC respectively), and have completely different features and implementations. For example, PX4 has 426 configurable parameters and Ardupilot has 612. The meanings of these parameters are quite different. From our experience, drone controlled by PX4 took much fewer mission time than the drone controlled by Ardupilot. However, simply tuning the flying speed parameter does not work because many parameters need to be tuned accordingly. For example, the

way-point radius represents a distance from a way-point, that when crossed means it has been hit. Increasing the speed of the drone without changing the way-point radius could result in overshoot, which makes the drone waste time to fly back to the track.

In order to achieve optimal flying performance, both PX4 and Ardupilot provide their own specific black-box parameter tuning tools. However, it allow tuning a very limited number of parameters 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, 41, 62]. `We aim to tune the parameters of Ardupilot to make it learn the flying behavior of PX4.`

We identify 20 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. For example, turning is achieved by lowering the speed of a subset of the 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 motors speed between the two controllers. Furthermore, as a typical mission in Ardupilot often needs to execute under multiple flight modes (e.g., takeoff, land, etc), we define the tuning regions as the individual mode control functions. In terms of implementation, we added only 22 library calls and one callback function with 204 LOC in the original Ardupilot controller.

To tune Ardupilot according to PX4's behavior, we first fly both Ardupilot and PX4 under 2 different missions. The first one is fairly simple, consisting of taking off, rising to 10 meters, and finally landing. The second mission is more complex, namely flying along a 45m route that contains 3 way points. Our experiments are conducted using the Gazebo simulator [35]. The first mission uses 1500 sample runs, while the second uses 4500 runs given its complexity, each taking 20-30 seconds. Overall, the tuning time is about 42 hours due to real-time simulation. We will explain why black-box tuning is unable to achieve the same goal later. Finally, to test the subsequent performance of Ardupilot, we fly it with the tuned parameters under a different and more complex test mission, in which the drone zigzags and returns to the starting point with a flight distance of 165m.

Fig. 24 shows both the motors speed and visual results for the first tuning mission. Note that motor speeds represent the key states of a drone. The gray point in the visual results indicates the front of the drone. As illustrated, PX4 first accelerates the drone to a high speed (with the initial spikes of motor speeds close to time 0). It then maintains a stable high speed till until timestamp 7 (sec). At this time, it decelerates as it reaches the targeted height. In contrast, the default Ardupilot

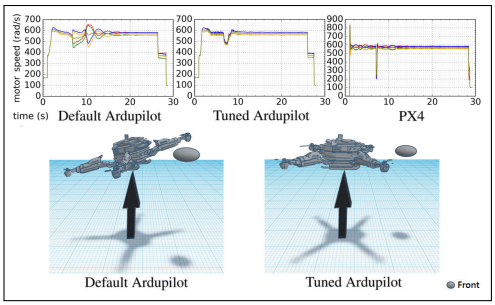


Figure 24: Tuning mission 1

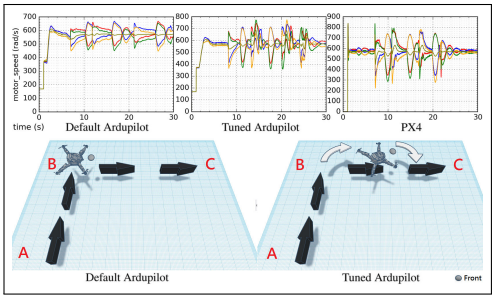


Figure 25: Tuning mission 2

rises very slowly and exhibits tilts and turns (due to some calibrations when taking off). After tuning, Ardupilot is more stable at take-off (i.e., the tilts and turns are avoided). If one looks into the motor speed chart, the spike and the dip (at 7th sec) appear, resembling the PX4's chart. While WBTuner is not able to achieve the same sharpness of the spike/dip as in PX4 due to other un-tuned parameters, the result is promising.

Fig. 25 shows the results of the second tuning mission (The three way points are indicated by A, B and C). When the default Ardupilot reaches the middle way point (i.e., B), it first tilts, turns at the same spot until its head points to the next way point C, and then flies towards C. Intuitively, changing the orientation at point B requires the drone to decrease its speed, and hence leads to a longer mission. Conversely, both PX4 and the tuned Ardupilot avoid turning as much as possible at point B, and rather change their orientation while flying towards C (as indicated by the white curved arrow), consequently finishing the mission in a much shorter time.

Fig. 26 shows the results of the new mission. Observe that 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% mission time decrease). The recorded videos for the test mission simulations are available at [1, 2, 47].

OpenTuner cannot be applied in this case 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

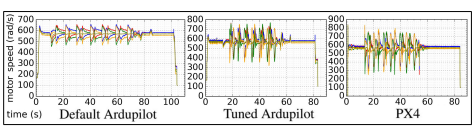


Figure 26: Testing mission

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 full execution). This is not a problem for WBTuner as it can spawn all the sampling/tuning processes after a successful start.

6. Related Work

There are several autotuning frameworks for domain-specific programs. For example, [13, 16] aimed to tune data-mining algorithms; [58] aimed to generate an optimized matrix multiply routine by empirical autotuning; [20] is specialized for tuning stencil computation; and [29] is a stochastic approach for parameter tuning of SVM. However, OpenTuner [7] provides a general autotuning framework that allows users to use multiple different tuning methods for different programs. Furthermore, OpenTuner treats the whole program as a black box. In contrast, WBTuner adopts white-box and hence complementary to OpenTuner. For users with the source code and certain level of domain knowledge, WBTuner allows him/her to compose highly sophisticated tuning schemes by accessing and manipulating the internal states.

A number of dynamic autotuning frameworks like [3, 8, 12, 14, 18, 28, 48, 50] have been proposed to monitor program execution to guide the program to perform self-adaptation for achieving specific optimization goal. For example, [18] provides a framework for tuning resource-aware distributed applications. PowerDial [28] transforms static application configuration parameters into dynamic controllable variables to make programs power-aware. However, these techniques mostly focus on tuning for a specific objective and for certain types of programs.

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. PetaBricks 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. It cannot tune pre-existing non-streaming programs where inter-dependences across phases are substantial like in Ardupilot. Furthermore, users need to use its language to write data processing programs.

Automatic parallelization [4, 54] transforms a sequential program to its concurrent version. The process is guided by annotations. Although parallelization spawns processes/threads, it divides a computation task into multiple concurrent subtasks. In contrast, WBTuner spawns processes to compute similar but different tasks.

7. 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 data processing programs. Our experiments show that WBTuner substantially improves data processing results and outperforms the state-of-the-art black-box tuning engine.

References

[1] Adrupilot-default. "https://drive.google.com/open?id=0BxgPTM7nEUyCcTFKdjM4RVNrMk0", 2017.

[2] Adrupilot-tuned. "https://drive.google.com/open?id=0BxgPTM7nEUyCYmVPdzBtMnoyT1U", 2017.

[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*, 2011.

[5] Olov Andersson, Mariusz Wzorek, and Patrick Doherty. Deep learning quadcopter control via risk-aware active learning. In *AAAI 2017*, 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*, 2009.

[7] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *PACT 2014*, 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*, 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*, 2011.

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

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

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

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

[14] 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*, 2006.

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

[16] Ole Burmeister, Markus Reischl, Georg Bretthauer, and Ralf Mikut. Data-mining-analysen mit der matlab-toolbox gait-cad (data mining analyses with the matlab toolbox gait-cad). *Automatisierungstechnik*, 2008.

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

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

[19] Hong Changa and Dit-Yan Yeung. Robust path-based spectral clustering. *Pattern Recognition*, 2008.

[20] 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*, 2011.

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

[22] 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.

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

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

[25] Limin Fu and Enzo Medico. Flame, a novel fuzzy clustering method for the analysis of dna microarray data. *BMC Bioinformatics*, 2007.

[26] A. Gionis, H. Mannila, and P. Tsaparas. Clustering aggregation. *ACM Trnsactions on Knowledge Discovery from Data*, 2007.

[27] 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.

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

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

[30] Anil Jain and Martin Law. Data Clustering: A User's Dilemma. 2005.

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

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

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

[34] Levente Kocsis and Csaba Szepesvári. Universal parameter optimisation in games based on spsa. *Machine Learning*, 2006.

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

[36] M. Lichman. UCI machine learning repository. ""http://archive.ics.uci.edu/ml", 2013.

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

[38] 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.

[39] Lorenz Meier. Px4 autopilot. "http://px4.io/", 2009.

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

[41] 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 28*, 2015.

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

[43] 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*, 2011.

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

[45] 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*, 2011.

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

[47] PX4. "https://drive.google.com/open?id=0BxgPTM7nEUyCYnRxS2FSN2JRbEE", 2017.

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

[49] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 1987.

[50] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 2009.

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

[52] James C. Spall. Feedback and weighting mechanisms for improving jacobian estimates in the adaptive simultaneous perturbation algorithm. *IEEE Transactions on Automatic Control*, 2009.

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

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

[55] C. Veenman, M. Reinders, and E. Backer. A maximum variance cluster algorithm. *Pattern Analysis and Machine Intelligence*, 2002.

[56] 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.

[57] WBTuner. Wbtuner source and tech report. "https://github.com/evans14641/WBTuner", 2017.

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

[59] Madhu Yedla, Srinivasa Pathakota, and T M Srinivasa. Enhancing k-means clustering algorithm with improved initial center. *International Journal of Computer Science and Information Technologies*, 2010.

[60] 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.

[61] C. T. Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computers*, 1971.

[62] T. Zhang, G. Kahn, S. Levine, and P. Abbeel. Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016.