

**Program Auto-tuning Through Population-based  
Stochastic Optimization**

by

Minshu Zhan

Submitted to the Department of Electrical Engineering  
and Computer Science

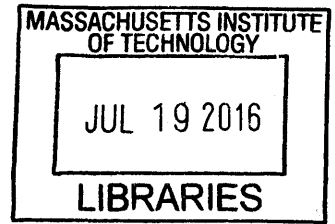
in partial fulfillment of the requirements for the degree of  
Master of Engineering in Electrical Engineering and Computer Science  
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2016

© Minshu Zhan, MMXVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and to  
distribute publicly paper and electronic copies of this thesis document  
in whole or in part in any medium now known or hereafter created.



**ARCHIVES**

**Signature redacted**

Author .....

Department of Electrical Engineering  
and Computer Science  
February 1, 2016

**Signature redacted**

Certified by .....

Kalyan Veeramachaneni  
Research Scientist  
Thesis Supervisor

**Signature redacted**

Certified by .....

Una May O'Reilly  
Principal Research Scientist  
Thesis Supervisor

**Signature redacted**

Accepted by .....

Dr. Christopher J. Terman  
Chairman, Masters of Engineering Thesis Committee



# Program Auto-tuning Through Population-based Stochastic Optimization

by

Minshu Zhan

Submitted to the Department of Electrical Engineering  
and Computer Science  
on February 1, 2016, in partial fulfillment of the  
requirements for the degree of  
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

Program optimization often can and need to be decoupled from the programs innate logic. In various domains, program autotuners have been developed which can automatically search for optimal program configurations using established optimization methods. OpenTuner is a general-purpose autotuner which provides a flexible search domain representation and robust optimization techniques. This thesis extends OpenTuner with the family of *population-based stochastic optimization (PBSO)* algorithms to boost OpenTuner's search capability on discrete-valued tuning problems.

Thesis Supervisor: Kalyan Veeramachaneni  
Title: Research Scientist

Thesis Supervisor: Una-May O'Reilly  
Title: Principal Research Scientist



## Acknowledgments

I would first like to express my sincerest appreciation towards my thesis supervisors, Prof. Una-O'Reilly, Dr. Kalyan Veeramachaneni, and Prof. Saman Amarasinghe for their persistent intellectual and emotional support throughout my research and study. Without their guidance and encouragements, completion of this thesis would not be possible.

My gratitude also goes to Dr. Jason Ansel from the Commit research group and Dr. Jonathan Ragan-Kelly from the Computer Graphics Group who provided resources and information crucial for my work.

I would also like to thank MIT EECS department for supporting my study financially. I am in particular grateful to Ms. Anne Hunter, administrator of the EECS M.Eng program, who helped me with many administrative issues.

Finally, I need to thank my family who have been always been understanding and supportive of me wherever I am, whatever I do.



# Chapter 1

## Introduction

Optimizing the performance of a program is a time-consuming process. Often the optimal configuration of a complex program under given conditions cannot be easily determined manually. Moreover, introduction of performance-boosting technologies such as parallelization creates additional tuning problems that can be beyond the expertise of the application developers.

To address this problem, many applications choose to abstract away performance-sensitive parameters and incorporate *program auto-tuners* to search for optimal values for these parameters. These auto-tuners are usually internal to the specific applications and adopt specific search methods.

OpenTuner is an application-general program auto-tuner that brings together these efforts of individual auto-tuner developments. It features a flexible problem representation scheme and a diverse bank of built-in search strategies.

The thesis focuses on enhancing OpenTuner’s search power on discrete-valued tuning problems. We achieve this mainly by introducing the *populated-based stochastic optimization (PBSO)* methods to OpenTuner. PBSO methods have shown good performances on discrete optimization problems found in other fields such as *operational research*. We perform tuning experiments on selected target programs and show that, depending on the type and size of the parameter space in question, different levels of performance boosts can be achieved,





# Chapter 2

## Related Works

### 2.1 Program Autotuning

Optimizing or *tuning* a computer program is not an easy task. To optimize, there are often numerous applicable algorithms and techniques to choose from, each of which is suitable for a particular use case, and each algorithm usually has a number of *parameters* that can be adjusted for the specific application. Furthermore, real-world applications are usually composed of many subprograms each of which can execute in different orders and which have different parameters with which to tune them. When the input is large, we have additional parameters to determine for parallelization, such as those which control granularity.

In the context of this thesis, *Configuring* a program means fixing the values of all the optimization parameters in the program, and the *configuration* of a program is the collection of these parameter values. Tuning a program by hand is usually very time-consuming and demands expertise that can be beyond the program's creator. For example, one researcher who specializes in signal processing does not necessarily know how to efficiently sort a big array using the machines she processes, yet optimizing the sort routine may give a big boost to her signal processing application and saves her precious research time.

Program *autotuners* are systems that automatically search for optimal configurations for the *target programs* - the programs that demand tuning - using estab-

lished optimization algorithms. One of the pioneer program autotuners is Halide, an autotuner-compiler that uses stochastic search to tune image processing programs. It finds optimal "schedules" for running image processing pipelines consisting of many stages and branches and achieves a tuning performance that surpasses hand-tuning with much less time [10]. Another autotuning system, Petabricks, is concerned with the situation where a program is composed of procedures each with multiple algorithmic choices as well as parameter values, which was shown able to achieve near-optimal performance[1].

Very recently, Jason et al proposed *OpenTuner* [2], an all-purpose program autotuner that can work on different application domains. As the foundation work for this thesis, it will be described briefly in the next section.

## 2.2 OpenTuner

OpenTuner has the following features:

**Flexible problem domain representation** OpenTuner is designed to be able to flexibly represent program configuration from different application domains.

**Ensemble technique** OpenTuner introduces bandit meta-techniques which allows several different optimization algorithms to to be explored and evaluated to favor using the one more likely to provide the best optimization. The meta-technique takes care to balance exploitation and exploration when applying the base techniques.

**Asynchronous search management** Both generation and evaluation of new program configurations occur asynchronously. This makes tuning efficient and parallelizable.

**Multi-objective search capability** Instead of optimizing for a single objective, OpenTuner can handle multiple objectives, e.g. minimizing execution time and maximizing some user-defined score at the same time.

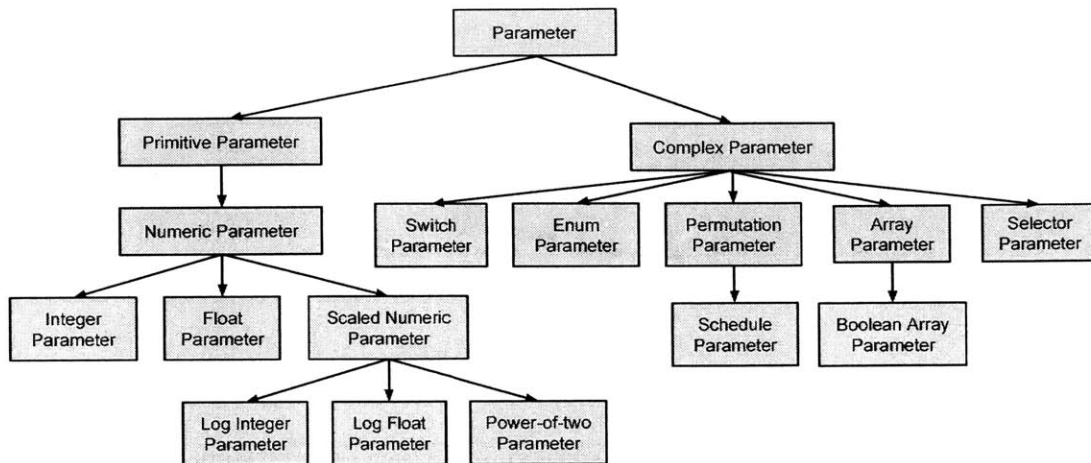


Figure 2-1: Existing parameter classes in OpenTuner

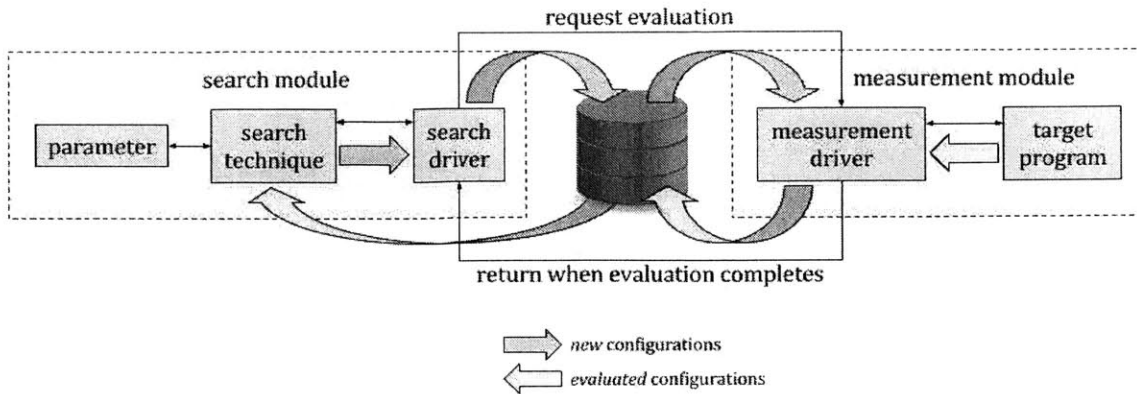
### 2.2.1 Problem space representation

In order to flexibly express the tuning problems from across different applications, OpenTuner provides users with a bank of elementary data types, or *parameters*, as shown by 2-1. Each *Parameter* class has by default *randomize* function and the numeric parameters have additional *linear operations* defined.

Tuning a target program involves interaction among three OpenTuner components - the search module, the measurement module, and the OpenTuner database. The search module is responsible for searching and creating new program configurations while the measurement module evaluates the configurations. The database serves as an archive, recording optimization attempt details.

### 2.2.2 Workflow

The tuning process of OpenTuner proceeds as follows. To begin, the search driver calls the search technique to obtain a new configuration. The search techniques are implemented in the form of *generators*, which means that, when a search technique finds a new solution, its optimization routine is paused and the technique returns control to its caller - the search driver. The search driver then labels the new configuration as “to-be-evaluated”, stores it in the OpenTuner database, and calls the measurement



**Figure 2-2:** Workflow of the original OpenTuner system.

driver for evaluation. The measurement driver retrieves from the database configurations that bear evaluation request labels, compiles the target program with these configurations, re-labels these configurations as “completed”; after the program has executed with the configuration, the measurement driver sends the performance scores back to the database. The search driver waits until the evaluation is completed. It then checks the convergence criteria. If convergence condition is not met, the previous steps are repeated - the search driver again calls the search technique; the technique resumes its optimization routine from where it stopped last time and runs until the next solution is created, etc.. Thus the process goes on. A schematic of the workflow is shown in figure 2-2.

The performance measure of a configuration is usually the execution time of the target program compiled under this configuration. If the execution takes too long, the program is terminated early and its execution time is set to infinite. When the search techniques needs to access the performance measure of an old solution, it retrieves it from the database.

### 2.2.3 Existing search techniques

All existing search techniques in OpenTuner are *iterative* optimization methods that continuously produce candidate solutions with increasing qualities. Existing techniques include simplex methods, differential evolution, simulated annealing, pattern

search, and random search.

In addition to specific optimization methods, the original OpenTuner also implements *meta-techniques* which simultaneously run several different optimization algorithms. In particular, the bandit meta-technique is an ensemble technique that let different base techniques take turns to run. This meta-technique dynamically adjusts priority of each base technique in the ensemble during the search according to its performance. The prioritized techniques will run more often to *exploit* its search capability, while the techniques that are not performing are let to run occasionally. The use of a bandit technique prevents the tuning from being stuck in local optimums that are created by running a single particular search algorithm.

## 2.3 Difficulty of complex tuning spaces

One particular difficulty of program auto-tuning is the complexity of the configuration search space. As a program's configuration involves many parameters of different types, the search space of program configuration is usually compound of different types of subspaces. Beside continuous and discrete numeric values, a program's configuration often involves ordinal choices, permutations, schedules, and graphs with even more complicated constraints. For examples, a simple *sort* routine can involve first choosing a sort algorithm that is suitable for the problem from available algorithms such as mergesort and quicksort. Many algorithms have numerical parameters that need to be further tuned. For example, when sorting a big array, brute-force algorithms may be more appropriate at lower levels, where the input is divided into small batches to be sorted individually; when all batches are sorted, mergesort can come in to sort the whole input array - the choice that we need to make is at which stage to change the search algorithm. Lastly, when the input is large and parallelization is required, the routine may need a *composition* of sub-algorithms, each of which has its own parameters to tune, thus the composition and sub-parameters needing to be optimized together in order to efficiently execute the task.

In image processing, for example, a program is often a pipeline of many different

tasks, some of which can be performed in parallel while some have dependencies on one another; furthermore, each individual task itself requires parallelization that need to be configured.

Most well-established gradient-based optimization techniques cannot be directly applied to solve these auto-tuning problems, if at all. On discrete spaces such as boolean, permutation, and graphs, the notion of *gradient* usually cannot be appropriately defined, or even approximated. The mixture of different spaces also makes it difficult to apply these methods.

The difficulty is partially handled by the original OpenTuner. Although no meaningful linear operations can be used on complex parameters, random perturbation and an appropriate selection scheme make it possible to search these discrete spaces. Pattern search is a basic algorithm that is derivative-free and has good convergence. First, a tentative solution is randomly initialized in the problem space. Then, at each iteration, new solutions are generated by "stepping away" from the old solutions at each dimension: if one of the new solutions outcompetes the old, it is passed on to the next generation; if not, the step size is shrunk and the iteration restarts [4]. Simplex methods - specifically Nelder-Mead method - works similarly except that a "simplex" of  $D + 1$  points instead of a single point is maintained throughout the search. The worst point in the old simplex is "reflected" across the centroid of the other vertices to generate a new vertex: if the new vertex is an improvement, it replaces the old vertex and the simplex expands towards the new vertex; otherwise, the simplex is shrunk [9]. In both of these methods, randomization replaces the neighbourhood motion (stepping and reflecting) in the case of complex parameters since linear operation is not defined.

Apart from being gradient-free, these methods are preferred because of their *iterative* property. In the case of program auto-tuning, near-optimal solutions produced within reasonable time limit is much more useful than exact optimums that take unacceptably long time to find.

However, the existing techniques still have the drawback that complex parameters cannot make meaningful movements in the search space, and the search can be

trapped in local optima easily.

A family of optimization methods that have proved to be effective in similar situations in operational research are the *population-based stochastic optimization* (PBSO) algorithms. Instead of a small set of candidate solutions, these algorithms maintain a large, diverse *population* of solutions to make the optimization robust. When generating new solutions, PBSO techniques introduces randomness which further prevents the problem of local optima.





# Chapter 3

## Design Proposal

### 3.1 Goals

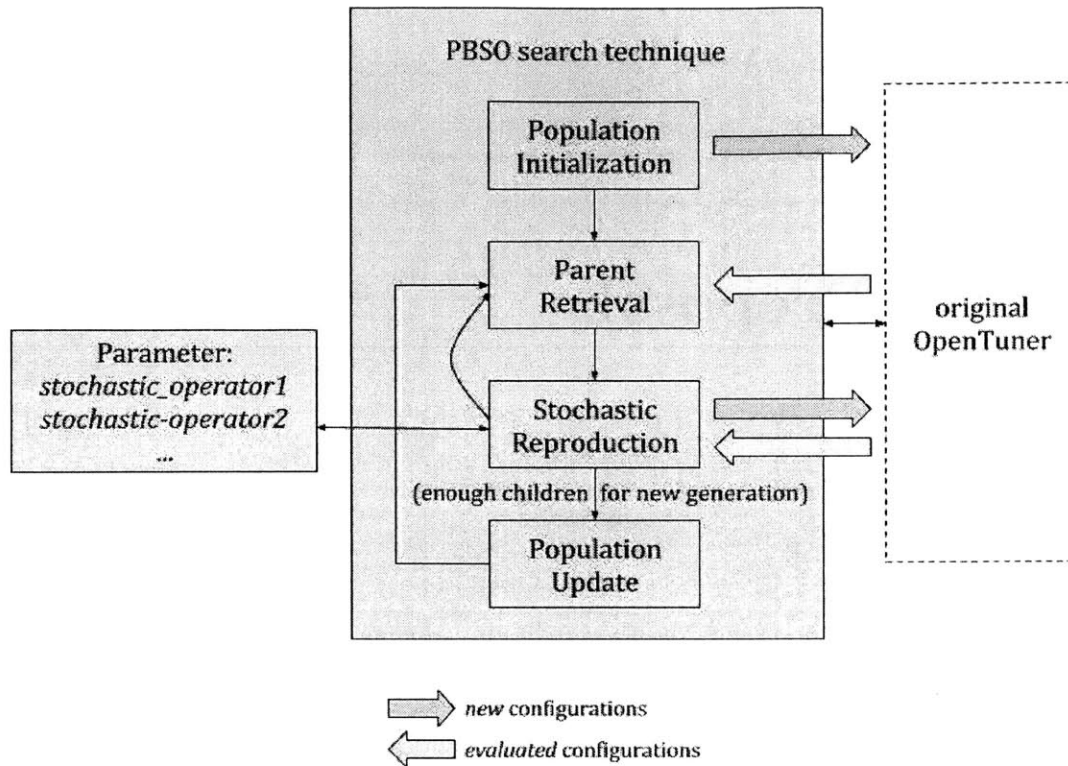
The primary goal of this thesis is to extend OpenTuner to enhance its searching capability on complicated domains. We do this by introducing *population-based stochastic search optimization* (PBSO) methods. Specifically, the extension improves OpenTuner’s tuning performance on integer, boolean, and permutation tuning domains.

Our secondary goal is to separate parameter-specific *stochastic operators* from the parameter-independent PBSO algorithm logics in our implementation. A PBSO algorithm should be able to easily switch between different stochastic operators, and a stochastic operator should be accessible by different PBSO algorithms. We should also be able to compose many variants of a PBSO algorithm by ”plugging in” different stochastic operators, and we should be able to modify or add PBSO techniques without reimplementing existing stochastic operators.

### 3.2 Overview

We implement the PBSO functionality for OpenTuner by extending the search module of the original OpenTuner. We create the `PBSOSearchTechnique` classes which inherit OpenTuner’s original `SearchTechnique` class, and we add *stochastic operators* as new methods of subclasses of the parameter class. A PBSO technique can be combined

with different stochastic operators to construct different technique variants.



**Figure 3-1:** The PBSO extension for OpenTuner. The `PBSOSearchTechnique` class inherits `SearchTechnique`. The parameter class is extended to include stochastic operators. See figure 2-2 for the original OpenTuner framework.

Figure 3-1 illustrates our PBSO extension. A PBSO technique maintains a collection of candidate solutions called the *population*. The optimization proceeds by iteratively improving the population. The technique first initializes the population, and then enters a loop consisting of three stages - parent retrieval, stochastic reproduction, and population update.

During parent retrieval, a small number of candidate solutions are taken out of the population as *parents*. The stochastic reproduction stage is composed of one or more stochastic operators, which recombine the parents and produce a *child* solution. After enough repetitions, the entire population is replaced by child candidate solutions and a new *generation* is created.

Every stochastic operator is originally innate to a particular PBSO algorithm. This extension decouples the stochastic operators from their original algorithms and share them among all PBSO techniques when compatible. On any given tuning problem, i.e. the involved parameters are specified, the compatibility between an parameter operator and a PBSO technique is determined solely by the number of parent candidates required by the technique’s reproduction stage. For example, the innate operator for PSO is the three-input operator **PSO-recombination** and for DE it is the three-input operator **DE-recombination**. Since the same number of parents are required, PSO can use **DE-recombine** instead of **DE-recombination** for its reproduction.

We will now describe in detail the PBSO techniques and stochastic operators implemented in the extension.

### 3.3 PBSO techniques

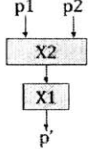
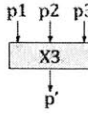
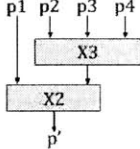
In this thesis we implement three representative PBSO algorithms, namely Genetic Algorithm (GA), Particle Swarm Optimization (PSO), and Differential Evolution (DE). The techniques are specified in terms of the four algorithmic stages as in figure 3-1, namely *population initialization*, *parent retrieval*, *stochastic reproduction*, and *population update*.

As described in 2.2.2, in the original OpenTuner every new candidate solution is evaluated right after it is produced, before the search technique proceeds to the next stages. Therefore, in the technique descriptions below, we assume that the performance score of each candidate solution is always known when the technique needs them.<sup>1</sup>

We provide a summary of the techniques in table 3.1.

---

<sup>1</sup>In the original OpenTuner, it is possible to parallelize the evaluation i.e. the search technique would yield a batch of candidate solution s instead of only one before requesting evaluation. In this extension, such parallelization is disabled, because some PBSO techniques need to access the scores of *all* existing candidate solution s when it produces a new candidate solution.

stage	GA	PSO	DE
Population Initiation	random	random	random
Parent Selection	$p1, p2$ are randomly selected from population based on fitness scores.	$p1$ is yielded by population iterator; $p2$ is the historical optimum of $p1$ ; $p3$ is the population's global optimum	$p1$ is yielded by population iterator; $p2, p3, p4$ are randomly selected from population
Stochastic Reproduction			
Population Update	The whole population is replaced when enough children are generated.	$p1$ is replaced by $p'$ .	$p1$ is replaced by $p'$ .

**Table 3.1:** Summary of the proposed PBSO techniques. X1, X2, X3 represents required operators which take one, two, and three parents respectively.

### 3.3.1 Genetic Algorithm

GA mimics the phenomena of biological evolution, where a community of organism evolve to become fitter under the pressure of natural selection.

#### Main algorithm

1. **Population Initiation** The initial population is created by randomly sampling the search domain.
2. **Parent Retrieval** Two parents  $p1, p2$  are randomly selected from the population based on their *fitness scores*. Specifically, we use the Stochastic Universal Sampling proposed by Baker for the parent selection, which ensures that higher-scored candidates are more likely to enter the new generation while leaving reasonable chance to less fit candidates. Fitness scores of candidate solutions are based on the *rankings* of their raw performance scores. Specifically, the fitness score of candidate  $i$  is  $1/\sqrt{r_i}$ , where  $r_i$  is the rank of  $i$  [3]. In practice this

is important for difficult tuning problems when the objective is to minimize the execution time of the tuned program. During the initial stage of tuning, most candidates lead to very slow programs that need to be killed before finishing, thus receiving 0 scores from the evaluator. Using the rank rather than the raw scores as the fitness measure prevents the population manager from eliminating these candidates prematurely during selection.

3. **Stochastic Reproduction** Two parents candidates  $p1, p2$  are processed by an two-input stochastic operator followed by a one-input stochastic operator to yield the child candidate solution  $p'$ .
4. **Population Update** The population is entirely replaced by child candidate solutions when the number of child candidates reaches the designated population size.

### 3.3.2 Particle Swarm Optimization

Particle Swarm Optimization (PSO) simulates the process that a swarm of particles seek for the globally optimal point in the search space. A candidate solution is represented by the position of a particle. In addition to the particle positions, PSO keeps track of these following states during the search:

$gbest$ : best candidate solution found by the entire population

$pbest_i$ : best candidate solution found by particle  $i$  for all  $i$

$v_i$ : (optional) velocity of particle  $i$  for all  $i$

#### Main algorithm

1. **Population Initiation** The population is initialized randomly.
2. **Parent Retrieval** A circulator population iterator<sup>2</sup> yields the particle indexed

---

<sup>2</sup>An iterator that passes through all members of the population in order and returns to the first member after it reaches the last.

- i*. The *primary* parent  $p1$  is set to be the position of particle  $i$ . Two *auxiliary* parents are set to be  $p2 = gbest$  and  $p3 = pbest_i$ .
- 3. **Stochastic Reproduction** A three-input stochastic operator transforms the three parents  $p1, p2, p3$  into a child candidate solution  $p'$ . Velocity of the particle  $i$  is updated optionally.
- 4. **Population Update** Every particle's position, i.e. the primary parent, is updated to the child candidate solution  $p'$  found in reproduction.

### 3.3.3 Differential Evolution

Differential evolution takes the idea of evolution and use the *differences* between population members to determine the degree of stochastic variation applied in reproduction.

#### Main Algorithm

- 1. **Population Initiation** The population is initialized randomly.
- 2. **Parent Retrieval** A primary parent  $p1$  is yielded by a circular population iterator as in PSO. In addition, three auxiliary parents  $p2, p3, p4$  are randomly picked from the population.
- 3. **Stochastic Reproduction** A three-input stochastic operator is first applied to  $p2, p3, p4$  to produce an intermediate child  $z$ . A two-input stochastic operator then transform  $p1$  and  $z$  into the final child  $p'$ .
- 4. **Population Update** Every primary parent  $p1$  is replaced by its child  $p'$  found in reproduction.

## 3.4 Stochastic Operators

In the stochastic reproduction step of PBSO techniques, we need stochastic operators that handle different parameter types. These stochastic operators are decoupled from

the main PBSO algorithms in our implementation.

In this extension we develop stochastic operators that operator on four parameter types, namely float, integer, boolean, and permutation. We focus on the latter three discrete types since our goal is to improve OpenTuner’s performance on discrete problem domains.

### 3.4.1 Float Operator

A parameter whose value is continuous has the following stochastic operators:

**F1 Additional noise** A scaled Gaussian noise is added to the parameter’s value.

**F2 2-way choice** One of the two parent float values is chosen at random.

**F3.1 3-way choice** One of the three parent float values is chosen at random.

**F3.2 PSO-recombination** Give constants  $c_1$ ,  $c_2$ ,  $c_3$  and velocity  $v$ , three parent float values  $p_1$ ,  $p_2$ ,  $p_3$  are recombined in the following manner:

$$v' = v * rand() * c_1 + (p_2 - p_1) * rand() * p_2 + (p_3 - p_1) * rand() * c_3 \quad (3.1)$$

$$p'_1 = p_1 + v' \quad (3.2)$$

where  $rand()$  generates a random float in the range  $[0, 1]$ . When applied in the original PSO algorithm, this operator updates the position of a particle from  $p_1$  to  $p'_1$  (equation (3.2)) and velocity from  $v$  to  $v'$  (equation (3.1)). When applied to other PBSO algorithms which do not maintain *velocities*, we simply set  $v = 0$  in equation (3.1).

**F3.3 DE-recombination** Given three candidate solutions  $p_1$ ,  $p_2$ ,  $p_3$ , this stochastic operator creates a mutant  $z$  by perturbing  $p_1$  with a force proportional to the difference between  $p_2$  and  $p_3$  as follows:

$$z = p_1 + F(p_2 - p_3) \quad (3.3)$$

where  $F$  is a scaling constant.

### 3.4.2 Boolean Operators

A parameter whose value is either 1 or 0 has the following stochastic operators:

**B1 Random flip** The parent boolean value is flipped with certain probability.

**B2 2-way choice** One of the two parent boolean values is chosen at random.

**B3.1 3-way choice** One of the three parent boolean values is chosen at random.

**B3.2 PSO-recombination** Kennedy and Eberhart adapted stochastic operator F3.2 for boolean values by modifying the position update equation (3.2) of operator F3.2 as:

$$x'_1 = \mathbb{1}[\text{rand}() < \sigma(v'_1)] \quad (3.4)$$

The velocity is transformed into a value in the range  $[0, 1]$  through the sigmoid function  $\sigma(x) = \frac{1}{1+e^{-x}}$  and used as a *probability* to determine the new particle position. The velocity update equation (3.1) remains the same [8].

**B3.3 DE-recombination** With a similar probabilistic interpretation as B3.2, we adapt stochastic operator F3.3 for the boolean domain as follows:

$$z = \begin{cases} p_1, & \text{if } p_2 = p_3, \\ f(p_1), & \text{if } p_2 \neq p_3 \end{cases} \quad (3.5)$$

where  $f$  is the random flip operator B1. This adaptation preserves the difference-based variation property of F3.3 in that  $p_1$  is altered only if  $p_2$  and  $p_3$  differ.

### 3.4.3 Integer Operators

A parameter whose value is an integer in the range of  $[M, N]$  has the following operators:



**I1 Additional noise** A scaled Gaussian noise is added to parent integer value  $p$ , followed by bounding and rounding:

$$n \sim N(p, \gamma(M - N - 1)) \quad (3.6)$$

$$p' = \begin{cases} M, & \text{if } n > M \\ N, & \text{if } n < N \\ \text{round}(n), & \text{if } N \leq n \leq M \end{cases} \quad (3.7)$$

**I2 2-way choice** One of the two parent integer values is chosen at random.

**I3.1 3-way choice** One of the three parent integer values is chosen at random.

**I3.2 PSO-recombination** Veeramachaneni adapted stochastic operator F3.3 for the bounded integer domain. Velocity update equation (3.1) remains the same. The original position update equation (3.2) is modified as

$$p' = f\left(\frac{M - N}{1 + e^{-v'}}\right) \quad (3.8)$$

where  $f$  is operator I1. This operator transforms the updated velocity  $v'$  into the range of  $[N, M]$  through a sigmoid function, perturbs it with a scaled Gaussian noise, and then sets it as the new particle position [11].

**I3.3 DE-recombination** We modify F3.3 by simply applying a scaled Gaussian noise:

$$z = f(p_1 + F(p_2 - p_3)) \quad (3.9)$$

where  $f$  is operator I1.

### 3.4.4 Permutation Operators

A parameter whose value is a permutation, i.e. an ordered list of unique elements, has the following operators:

- P1.1 Shuffle** The whole permutation list is shuffled.
- P1.2 Random swap** Two random elements of the permutation are swapped.
- P1.3 Random invert** Treating the permutation as a path, a randomly chosen subpath is inverted.
- P2.1 2-way choice** One of the two parent permutations is chosen at random.
- P2.2 Order Crossover** The child first inherits a random subpath from one parent; the elements outside the subpath are then arranged according to their orders in the other parent [6] [5].
- P2.3 Permutation Crossover** The elements in a random subpath of one parent is re-arranged according to their order in other parent [12].
- P2.4 Partially-Mapped Crossover** Two random cut points are first selected on the permutation array. Subpaths between the cut points from the two parents form a mapping. The parents then exchange the subpaths, creating two children. If one child is valid, i.e. the permutation does not contain repeated elements, it is returned. Otherwise, each repeated element is substituted by the element that it is mapped to in the subpath mapping [7].
- P3.1 3-way choice** One of the three parents permutations is chosen at random.
- P3.2 Conditional Order Crossover** This operator is constructed from P2.2 as follows:
- $$p' = \begin{cases} p_1, & \text{if } r \leq c_1 \\ p_1 \otimes p_2, & \text{if } c_1 < r \leq c_1 + c_2 \\ p_1 \otimes p_3, & \text{if } c_1 + c_2 < r < c_1 + c_2 + c_3 \end{cases} \quad (3.10)$$
- where constants  $c_1, c_2, c_3$  are normalized to 1,  $r = rand()$ , and  $\otimes$  represents P2.2.
- P3.3 Conditional Permutation Crossover** This operator is also constructed using equation (3.10), except that  $\otimes$  represent P2.3.

PBSO technique	required operator	parameter-specific operator choices			
		float	boolean	integer	permutation
GA	X2	F2	B2	I2	P2.1 - P2.4
	X1	F1	B1	I1	P1.1 - P1.3
PSO	X3	F3.1 - F3.3	B3.1 - B3.3.	I3.1 - I3.3	P3.1 - P3.4
DE	X3	F3.1 - F.3.3	B3.1 - B3.3.	I3.1 - I3.3	P3.1 - P3.4
	X2	F2	B2	I2	P2.1 - P2.4

**Table 3.2:** List of PBSO technique variants constructed by applying different operators for each parameter. X1, X2, X3 represents required operators which take one, two, and three parents respectively.

**P3.4 Conditional Partially-mapped Crossover** This operator is also constructed using equation (3.10), except that  $\otimes$  represent P2.4.

### 3.5 Construct PBSO Technique Variants

As stated before, a PBSO technique requires one or more stochastic operators during its stochastic reproduction stage. In this extension, we enable *operator sharing* among PBSO techniques, which means that a stochastic operator innate to a certain algorithm can be applied to others as long as the number of required parents agrees. For example, operator P2.x or the crossover operators are designed for GA but can be used by DE as well. Thus we can create a large set of PBSO technique variants by combining different techniques with different stochastic operators. Table 3.2 enumerates possible PBSO variants for each of the four parameter types.

When tuning a configuration that contains only one type of parameters, the number of variants for a PBSO technique is the *product* of the number of choices for all its required operators. For a pure float-valued configuration, for example, GA has only one choice for each of its required operators (F2 and F1), which multiply to 1 GA variant. On the other hand, for a pure permutation-valued problem, DE has 4 choices for its three-input operator (P3.1-P3.4) and 4 choices for the two-input operator (P2.1-P2.4), which multiply to 16 total DE variants. See table 3.3 for a full list of PBSO variant counts.

In practice, the configuration in question is usually a mix of different parameter

PBSO technique	float	boolean	integer	permutation
GA	1	1	1	12
PSO	3	3	3	4
DE	3	3	3	16

**Table 3.3:** Counts of PBSO variants for each parameter type.

types. In this case, a PSBO technique would need to select its operators for every involved type, thus the number of variants it could have becoming the product of its variants for individual types. For example, to tune a configuration that contains only boolean and permutation parameters, we have  $1 \times 12 = 12$  GA variants,  $3 \times 4 = 12$  PSO variants, and  $3 = 48$  DE variants.

## 3.6 Extending PBSO Techniques

### 3.6.1 Adding a PBSO technique

A new PBSO search technique should inherit `PBSOSearchTechnique` class and implements the following methods:

1. `init_pop` returns an population. The returned value is used as the initial population.
2. `get_parents` given the population, return a set of candidate solutions. The returned value is used as parents.
3. `update_pop` given a number of candidate solution and the population, update the population.

In addition, the class field `reproduction_ops` should be set to specify the stochastic operators required by the new technique.

### 3.6.2 Adding a stochastic operator

The stochastic operators are implemented as class methods of the parameter classes. To differentiate from other methods in the class, a stochastic operator must bear the

identifying pre-fix `op_` in the function name to allow detection of the new operator. The only required inputs for the stochastic operator should be the parent parameters, the number of which is used to determine operator-technique compatibility.



# Chapter 4

## Experiment

### 4.1 Test Problem: Halide

To demonstrate and analyze the power of PBSO techniques in program tuning, we test the PBSO-extended OpenTuner on Halide problems which are concerned with tuning image processing applications.

An image processing pipeline is a set of interconnected computations performed on image data for image processing purposes such as smoothing, compression, and pattern recognition. Such code is often hard to optimize by hand due to complicated tradeoffs between redundant computation and data storage. The specific test application we choose is the bilateral filter program, which smooths an image while preserving major boundaries.

The application's search domain is described as follows:

- 38 integers (`PowerOfTwoParameters` with range  $1-2^3$ ) which mostly determine granularity of parallelization in the pipeline,
- 16 booleans which dictate whether parallelization or order storage is used, and
- 17 permutation-based variables which decide the order of executing specific tasks, which includes:
  - 8 small `PermutationParameters` (length=2,3,4)

- 8 schedule ScheduleParameters (length=4,6,8)
- 1 big HalideComputeAtScheduleParameter (length=60)

## 4.2 Experiment Setup

We divide the search space into three subspaces by type: the *integer* subspace, the *boolean* subspace, and *permutation* subspace. Tests are performed on these subspaces individually as well as the on the whole space. When a subspace is tested, the value of the rest of the domain is fixed to that from a randomly-generated feasible solution. We divide the configuration search space for two purposes: first, for a search space as large as a Halide program, searching on subspaces separately may be necessary; second, the divided experiment help illustrate properties of different search spaces and how different search techniques work on them, which potentially guide us to design useful higher-level search strategies.

We use PSO, GA, DE, and a bandit technique incorporating the three PBSO techniques. The population size is set to 30 for the three techniques. An additional pure random search is introduced as a basis for comparison. Each tuning run last 2 hours or until the technique consistently fails to generate an improved configuration.. Each technique-subspace combination is run 10 times independently.

## 4.3 Results

Each time we run a search technique, we obtain a performance curve where every point on the curve represents the target program's execution time compiled with the *best* configuration found by the technique up to that time. All performance curves are therefore non-decreasing. Each technique is run 10 times, for all search subspaces. We show the means over the 10 runs in figure 4-1 and 4-2. We use these curves to analyze the convergence behaviors of PBSO technique with respect to different subspaces.

In addition, we use the last point of each mean performance curve to calculate



	integer	permutation	boolean	mixed-type
PSO	28.6%	25.6%	32.5%	140.3%
DE	28.9%	-17.0%	3.4%	77.1%
GA	103.5%	58.0%	31.5%	309.2%
bandit	146.4%	11.4%	5.7%	34.4%

**Table 4.1:** Program speedup introduced by different PBSO techniques, compared with random search.

the final performance boost created by the corresponding PBSO method as compared against the baseline random search method. Table 4.1 shows the performance boost for each method-subspace as the percentage of execution speed (inverse of execution time) increased from the baseline method to the PBSO method.

### 4.3.1 Final Performance

Table 4.1 shows that, to different degrees, most PBSO methods perform better than random search. GA and PSO achieve significant speedup (great than 100%, i.e. half execution time) on the complete problem space. GA achieves the best final performance on the complete problem space (309.2%) as well as on the individual subspaces. The bandit technique composed of PSO, DE, and GA works well on the integer subspace, but does not create significant performance boost on the complete problem space. Both DE and the bandit technique appear to have trouble with boolean and permutation spaces, showing little speedup ( $< 10\%$ ). This means that, the permutation and boolean operators proposed in 3.4.4 and 3.4.3 may be problematic. In particular, the composed crossover implementation is likely not sufficiently representing the concept of “difference” to realize the functionality of DE.

Next we inspect all mean performance curves to further inspect how different search methods reach their final results.

## 4.3.2 Convergence-based Observation

### Full Tuning Space

By observing how the techniques perform on the full tuning problem space, as shown in figure 4-1a, we find that:

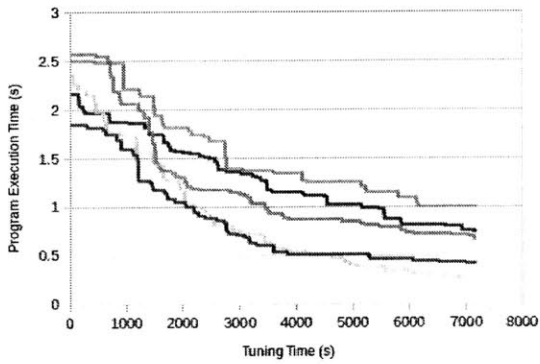
- First, none of the techniques has fully converged by the end of the tuning. This means that the search space has not been exhausted and that, given more tuning time, all techniques are likely to improve further.
- Overtime, all PBSO techniques perform consistently better than the baseline random search method. In addition, comparing the best performing methods, GA and PSO, GA demonstrates more potential on further improvement at the end of tuning with a sharper slope.
- The bandit technique does not reach the best performance achieved by its base techniques, given the same amount of tuning time. This is understandable, since as the ensemble technique has no prior knowledge on which technique works the best and needs to spend more time on exploration. However, as with GA, the slope at the end of the performance curve suggests large potential for improvement.

### Boolean

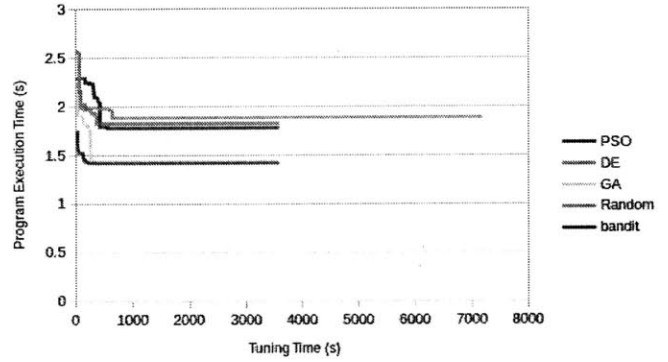
On the boolean subspace, as shown by 4-1b, all techniques stop improving after the first 10 minutes. This is most likely because of the small size of the subspace has been exhausted.

### Permutation

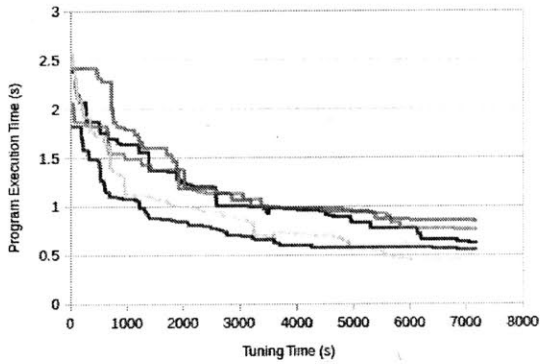
From 4-1c we see that, on the permutation subspace, performances of PSO and GA are similar to that on the full space, but their improvements happen earlier and more drastically and both appear close to convergence. This demonstrates the usefulness



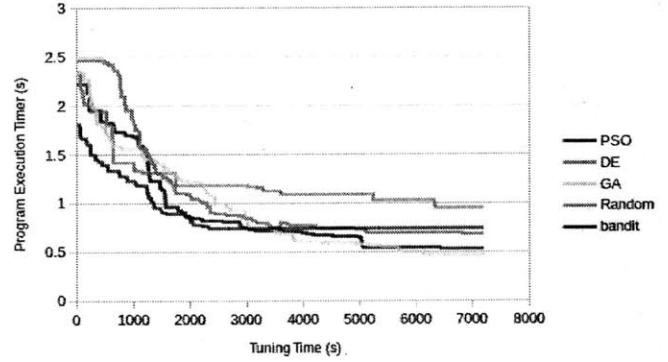
(a) Search on the entire problem space



(b) Search on the boolean subspace

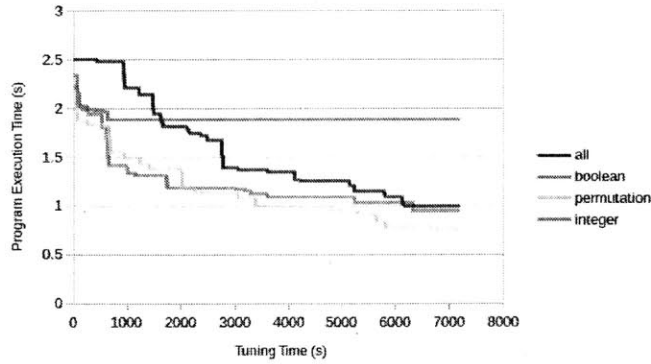


(c) Search on the permutation subspace

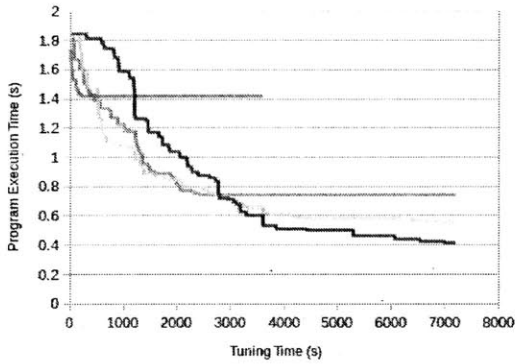


(d) Search on the integer subspace

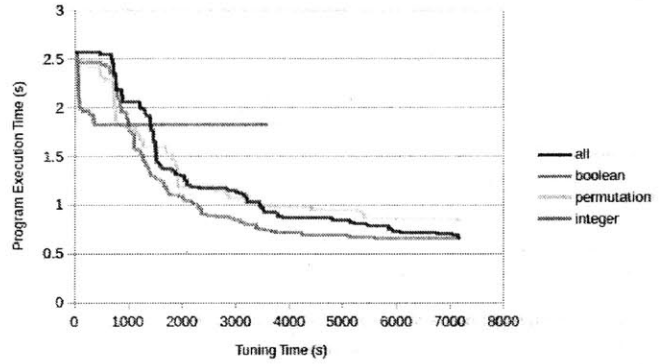
**Figure 4-1:** Auto-tuning a Halide program with different search techniques.



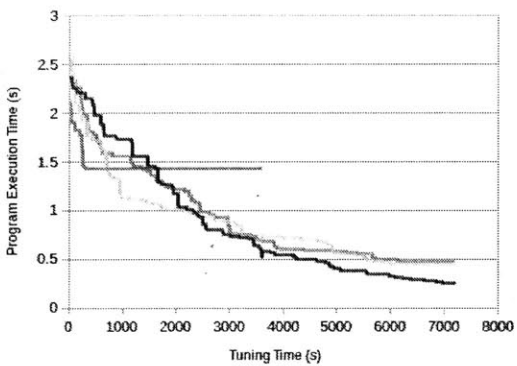
(a) Random search



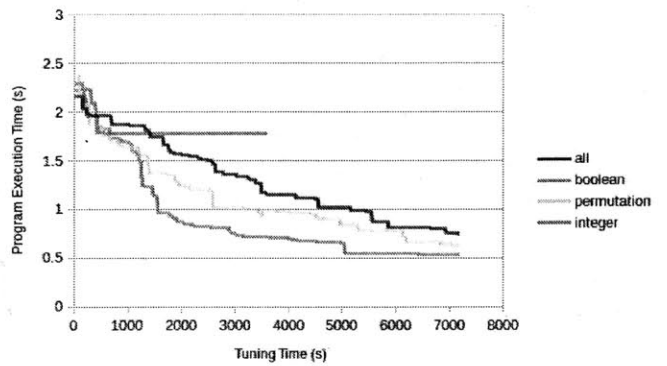
(b) Particle Swarm Optimization



(c) Differential Evolution



(d) Genetic Algorithm



(e) Bandit technique incorporating PSO, DE, and GA

Figure 4-2: Auto-tuning a Halide program on different subspaces

of the permutation operators. Unlike PSO and GA, DE turned out to behave almost like random search. One explanation can be that, unlike PSO and GA, DE does not directly use existing good solutions when generating new ones. It can also be caused by the fact that the 3-input permutation operator can not reflect the concept of “difference” which is essential to the performance of DE. The bandit technique does not reach the best performance among its base techniques and has not converged in the end, which indicates that the permutation subspace may still be too large for the ensemble technique to search within the given time constraint.

### **Integer**

The integer subspace is smaller than the permutation subspaces. As expected, convergence is more obvious than permutation subspace or the full space overall. In particular the bandit technique is able to converge and eventually reaches the best performance among its base techniques (GA). This verifies that, when given enough tuning time, the bandit technique is capable of utilizing all its base technique.

### **Method-independent Subspace Properties**

To better understand how search space type affect tuning, we re-plot the performance curves in groups of techniques, shown by figure 4-2.

The most important conclusion we can derive is that, tuning a *subspace* rather than the full mixed-type problem space can speed up the initial tuning phase, regardless of what search method is chosen. This suggests that subspace-based optimization strategies may improve tuning efficiency.



# Chapter 5

## Conclusions

In this thesis we introduce the family of *population-based stochastic optimization* methods to domain-generic program auto-tuner *OpenTuner* that enhances its tuning power on particular discrete problem spaces. The new functionalities are tested on the real tuning problem *Halide* and our major findings are as follows:

- Overall GA achieves the best tuning performance on all parameter types in concern, with a performance boost of up to 309.2% compared with random search, and shows the strongest potential for further performance boost given more tuning time.
- Auto-tuning can be accelerated by searching on subspaces of the problem domain instead of the complete search space.
- Given limited tuning time, a bandit technique may not be able to achieve the best results can be possibly found by its constituent methods.
- Most proposed discrete-domain adaptations for PBSO methods, realized as parameter-specific stochastic operators, work well on the test problem; however, the composed crossover operators (P3.X and B3.X) can have trouble when applied to DE.

In accordance with our findings, we recommend the following next-steps to improve the PBSO functionality in OpenTuner:

**Operator Re-design** The results for DE on the permutation and boolean problem domains suggests that the crossover-based interpretation of subtraction operation on the permutation space might not be sufficient for DE to work and need to be re-designed.

**Subspace search strategies** The acceleration effect of subspace search suggests that OpenTuner can benefit from incorporating configuration-level search strategies that divides the full problem space into subspaces and focus on different subspaces at different times.

**Self-tuning** OpenTuner's bandit meta-technique makes it possible to run multiple search techniques at the same time and automatically allow the good techniques to stand out. However, the exploration comes at the price of spending extra tuning time, and we should not completely rely on the meta-technique when it comes to choosing and configuring the right search algorithm. More experiments need to be done to determine reasonable default settings for specific search techniques, as well as choices of default techniques in an bandit technique given the tuning problem. For example, GA which appears to be more successful on discrete optimization than other methods may be given a higher initial priority in the ensemble technique it constitutes.



# Bibliography

- [1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM, 2009.
- [2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316. ACM, 2014.
- [3] James E Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the second international conference on genetic algorithms*, pages 14–21, 1987.
- [4] William C Davidon. Variable metric method for minimization. *SIAM Journal on Optimization*, 1(1):1–17, 1991.
- [5] Lawrence Davis. Job shop scheduling with genetic algorithms. In *Proceedings of an international conference on genetic algorithms and their applications*, volume 140. Carnegie-Mellon University Pittsburgh, PA, 1985.
- [6] Kusum Deep and Hadush Mebrahtu Adane. New variations of order crossover for travelling salesman problem. *IJCOPI*, 2(1):2–13, 2011.
- [7] David E Goldberg and Robert Lingle. Alleles, loci, and the traveling salesman problem. In *Proceedings of the first international conference on genetic algorithms and their applications*, pages 154–159. Lawrence Erlbaum Associates, Publishers, 1985.
- [8] James Kennedy and Russell C Eberhart. A discrete binary version of the particle swarm algorithm. In *Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on*, volume 5, pages 4104–4108. IEEE, 1997.
- [9] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

- [10] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.
- [11] Kalyan Veeramachaneni, Lisa Osadciw, and Ganapathi Kamath. Probabilistically driven particle swarms for optimization of multi valued discrete problems: design and analysis. In *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, pages 141–149. IEEE, 2007.
- [12] Darrell Whitley, Doug Hains, and Adele Howe. A hybrid genetic algorithm for the traveling salesman problem using generalized partition crossover. In *Parallel Problem Solving from Nature, PPSN XI*, pages 566–575. Springer, 2010.