Hyperparameter Tuning in Bandit-Based Adaptive Operator Selection

Maciej Pacula, Jason Ansel, Saman Amarasinghe, and Una-May O'Reilly

CSAIL, Massachusetts Institute of Technology, Cambridge, MA 02139, USA
{mpacula, jansel, saman, unamay}@csail.mit.edu

Abstract. We are using bandit-based adaptive operator selection while autotuning parallel computer programs. The autotuning, which uses evolutionary algorithm-based stochastic sampling, takes place over an extended duration and occurs *in situ* as programs execute. The environment or context during tuning is either largely *static* in one scenario or *dynamic* in another. We rely upon adaptive operator selection to dynamically generate worthy test configurations of the program. In this paper, we study how the choice of hyperparameters, which control the trade-off between exploration and exploitation, affects the effectiveness of adaptive operator selection which in turn affects the performance of the autotuner. We show that while the optimal assignment of hyperparameters varies greatly between different benchmarks, there exists a single assignment, for a context, of hyperparameters that performs well regardless of the program being tuned.

1 Introduction

We are developing an autotuning technique, called SiblingRivalry, based upon an evolutionary algorithm (EA) which tunes poly-algorithms to run efficiently when written in a new programming language we have designed. The autotuner runs in two different kinds of computing environments: static or dynamic. In either environment, multiple execution times and accuracy of results will vary to different degrees. Using special software infrastructure, the online technique, embedded and running in the run-time system, is able to continuously test candidate poly-algorithm configurations in parallel with the best configuration to date whenever a program is invoked. The technique generates a candidate configuration by selecting one of a set of specific mutation operators that have been derived for it during the program's compilation. If it finds a better configuration, it makes a substitution and continues. We call this process "racing". The technique needs to generate candidate configurations that both explore poly-algorithm space and exploit its knowledge of its best configuration.

The choice of which mutation operator to use is vital in optimizing the overall performance of the autotuner, both in time to converge to efficient programs and their answer quality. Some mutation operators will have large effects on program performance, while others will have little or no effect. If the evolutionary algorithm spends too much time exploring different mutation operators, convergence will be slow. If the evolutionary algorithm spends too much time

trying to exploit mutation operators that have yielded performance gains in the past, it may not find configurations that can only be reached through mutation operators that are not sufficiently tested. Noise in program performance due to execution complicates the picture and make optimal mutation operator selection imperative.

To address this challenge, SiblingRivalry uses what we call "bandit-based adaptive operator selection". Its underlying algorithm is the Upper Confidence Bound (UCB) algorithm, which is a technique inspired by a provably optimal solution to the Multi-Armed Bandit (MAB) problem. This technique introduces two hyperparameters: W - the length of the history window, and C - the balance point between exploration and exploitation. UCB is only optimal if these hyperparamters are set by an oracle or through some other search technique. In practice, a user of this technique must either use a fixed, non-optimal assignment of these hyperparameters, or perform a search over hyperparameters whenever the search space changes. Unfortunately, in practice, finding good values of these hyperparameters may be more expensive that the actual search itself. While [5] addresses the robustness of hyperparameters in empirical academic study, in this paper, we present a practically motivated, real world study on setting hyperparameters. We define evaluation metrics that can be used in score functions that appropriately gauge the autotuner's performance in either a static or dynamic environment and use them to ask:

- How much does the optimal assignment of hyperparameters vary when tuning different programs in two classes of environments - static or dynamic?
- Does there exist a single "robust" assignment of hyperparameters for a context that performs close to optimal across all benchmarks?

The paper proceeds as follows: in Section 2 we provide a necessarily brief description of our programming language and its autotuner. Section 3 reviews related work. Section 4 describes the UCB algorithm and the hyper parameters. Section 5 describes our evaluation metrics and scoring functions for tuning the hyperparameters. Section 6 provides experimental results. Section 7 concludes.

2 PetaBricks and its Autotuner

PetaBricks is a language designed specifically to allow the programmer to expose both explicit and implicit choices to an integrated autotuning system [1, 2]. The goal of the PetaBricks autotuner is to, on each machine, find a program that satisfies the user's accuracy requirements while minimizing execution time. Accuracy is a programmer-defined metric, while execution time is measured by running the program on the given hardware. Given a program, execution platform and input size, the autotuner must identify an ideal configuration which is a set of algorithmic choice and cutoff selectors, synthetic functions for input size transforms and a set of discrete tunable parameters. The autotuner is an evolutionary algorithm which uses a program-specific set of mutation operators.

These mutation operators, generated by the compiler, each target a specific single or a set of tunable variables of the program that collectively form the genome. For example, one mutation operator can randomly change the scheduling policy for a specific parallel region of code. Another set of mutation operators can randomly add, remove, or change nodes (one mutation operator for each action) in a decision tree used to dynamically switch between entirely different algorithms provided by the user.

3 Related Work and Discussion

In the context of methods in evolutionary algorithms that provide parameter adjustment or configuration, the taxonomy of Eiben [4] distinguishes between offline "parameter tuning" and online "parameter control". Operator selection is similar to parameter control because it is online. However, it differs from parameter control because the means of choosing among a set of operators contrasts to refining a scalar parameter value.

Adaptive methods, in contrast to self-adaptive methods, explicitly use isolated feedback about past performance of an operator to guide how a parameter is updated. An adaptive operator strategy has two components: operator credit assignment and an operator selection rule. The *credit assignment* component assigns a weight to an operator based on its past performance. An operator's performance is generally measured in terms related to the objective quality of the candidate solutions it has generated. The *operator selection rule* is a procedure for choosing one operator among the eligible set based upon the weight of each. There are three popular adaptive methods: probability matching, adaptive pursuit and multi-armed bandit. Fialho has authored (in collaboration with assorted others) a large body of work on adaptive operation selection, see, for example, [5, 6]. The strategy we implement is multi-armed bandit with AUC credit assignment. This strategy is comparison-based and hence invariant to the scale of the fitness function which can vary significantly between PetaBricks programs. The invariance is important to the feasibility of hyperparameter selection on a general, rather than a per-program, basis.

There is one evolutionary algorithm, differential evolution [10], that takes a comparison-based approach to search like our autotuner. However, differential evolution compares a parent to its offspring, while our algorithm is not always competing parent and offspring. The current best solution is one contestant in the competition and its competitor is not necessarily its offspring. Differential evolution also uses a method different from applying program-dependent mutation operators to generate its offspring.

4 Adaptive Operator Selection

Selecting optimal mutators online, while a program executes numerous times over an extended duration, can be viewed as an instance of the Multi-Armed Bandit problem (MAB), with the caveats described in [8]. We would like to

explore the efficacy of all mutators so that we can make an informed selection of one of them. The MAB resolves the need to optimally balance exploration and exploitation in a way that maximizes the cumulative outcome of the system.

In the general case, each variation operator corresponds to one of N arms, where selecting i-th arm results in a reward with probability p_i, and no reward with probability $1 - p_i$. A MAB algorithm decides when to select each arm in order to maximize the cumulative reward over time [8]. A simple and provably optimal MAB algorithm is the *Upper Confidence Bound (UCB)* algorithm, originally proposed by Auer *et al.* [3]. The empirical performance of the UCB algorithm has been evaluated on a number of standard GA benchmarks, and has been shown to be superior to alternative adaptive operator selection techniques such as Probability Matching [8].

The UCB algorithm selects operators according to the following formula:

$$\text{Select } \arg\max_i \left(\hat{q}_{i,t} + C \sqrt{\frac{2 \log \sum_k n_{k,t}}{n_{i,t}}} \right) \tag{1}$$

where $\hat{q}_{i,t}$ denotes the empirical quality of the i-th operator at time t (exploitation term), $n_{i,t}$ the number of times the operator has been used so far during a sliding time window of length W (the right term corresponding to the exploration term), and C is a user defined constant that controls the balance between exploration and exploitation [3, 8]. To avoid dividing by zero in the denominator, we initially cycle through and apply each operator once before using the UCB formula, ensuring $n_{i,t} \geq 1$.

Our PetaBricks autotuner uses the *Area Under the Receiving Operator Curve (AUC)* to compute the empirical quality of an operator. AUC is a comparison-based credit assignment strategy devised by Fialho et al. in [7]. Instead of relying on absolute average delta fitness, this method ranks candidates generated by a mutator i, and uses the rankings to define the mutator's *Receiving Operator Curve*, the area under which is used as the empirical quality term $\hat{q}_{i,t}$ (Equation 1). To extend this method to variable accuracy, we use the following strategy: If the last candidate's accuracy is below the target, candidates are ranked by accuracy. Otherwise, candidates are ranked by throughput (inverse of time).

5 Tuning the Tuner

The hyperparameters C (exploration/exploitation trade-off) and W (window size) can have a significant impact on the efficacy of SiblingRivalry. For example, if C is set too high, it might dominate the exploitation term and all operators will be applied approximately uniformly, regardless of their past performance. If, on the other hand, C is set too low, it will be dominated by the exploitation term $\hat{q}_{i,t}$ and new, possibly better operators will rarely be applied in favor of operators which made only marginal improvements in the past.

The problem is further complicated by the fact that the optimal balance between exploration and exploitation is highly problem-dependent [5]. For example, programs with a lot of algorithmic choices are likely to benefit from a

high exploration rate. This is because algorithmic changes create discontinuities in the program's fitness, and operator weights calculated for a given set of algorithms will not be accurate when those algorithms suddenly change. When such changes occur, exploration should become the dominant behavior. For other programs, e.g. those where only a few mutators improve performance, sacrificing exploration in favor of exploitation might be optimal. This is especially true for programs with few algorithmic choices - once the optimal algorithmic choices have been made, the autotuner should focus on adjusting cutoffs and tunables using an exploitative strategy with a comparatively low C.

The optimal value of C is also closely tied to the optimal value of W, which controls the size of the history window. The autotuner looks at operator applications in the past W races, and uses the outcome of those applications to assign a quality score to each operator. This is based on the assumption that an operator's past performance is a predictor of its future performance, which may not always be true. For example, changes in algorithms can create discontinuities in the fitness landscape, making past operator performance largely irrelevant. However, if W is large, this past performance will still be taken into account for quite some time. In such situations, a small W might be preferred.

Furthermore, optimal values of C and W are not independent. Due to the way $\hat{q}_{i,t}$ is computed, the value of the exploitation term grows with W. Thus by changing W, which superficially controls only the size of the history window, one might accidentally alter the exploration/exploitation balance. For this reason, C and W should be tuned together.

5.1 Evaluation metrics

Because there is no single metric that will suffice to evaluate performance under different hyperparameter values, we use three separate metrics to evaluate SiblingRivalry on a given benchmark program with different hyperparameters:

1. **Mean throughput**: the number of requests processed per second, averaged over the entire duration of the run. Equal to the average number of races per second.
2. **Best candidate throughput**: inverse of the runtime of the fastest candidate found during the duration of the run. For variable accuracy benchmarks, only candidates that met the accuracy target are considered.
3. **Time to convergence**: number of races until a candidate has been found that has a throughput within 5% of the best candidate for the given run. For variable accuracy benchmarks, only candidates that met the accuracy target are considered.

To enable a fair comparison between SiblingRivalry's performance under different hyperparameter values, we define a single objective metric for each scenario that combines one or more of the metrics outlined above. We call this metric the *score function* f_b for each benchmark b, and its output the *score*.

We consider two classes of execution contexts: static and dynamic. In the static context, the program's execution environment is mostly unchanging. In

this setting, the user cares mostly about the quality of the best candidate. Convergence time is of little concern, as the autotuner only has to learn once and then adapt very infrequently. For the sake of comparison, we assume in this scenario the user assigns a weight of 80% to the best candidate's throughput, and only 20% to the convergence time. Hence the score function for the static context:

$$f_b(C, W) = 0.8 \times \text{best\_throughput}_b(C, W) + 0.2 \times \text{convergence\_time}_b^{-1}(C, W)$$

In the dynamic context, the user cares both about average throughput and the convergence time. The convergence time is a major consideration since execution conditions change often in a dynamic system and necessitate frequent adaptation. Ideally, the autotuner would converge very quickly to a very fast configuration. However, the user is willing sacrifice some of the speed for improved convergence time. We can capture this notion using the following score function:

$$f_b(C, W) = 0.5 \times \text{mean\_throughput}_b(C, W) + 0.5 \times \text{convergence\_time}_b^{-1}(C, W)$$

We normalize throughput and convergence time with respect to their best measured values for the benchmark, so that the computed scores assume values in the range $[0, 1]$, from worst to best. Note that those are theoretical bounds: in practice it is often impossible to simultaneously maximize both throughput and convergence time.

6 Experimental Results

We evaluated the hyperparameter sensitivity of SiblingRivalry by running the autotuner on a set of four benchmarks: Sort, Bin Packing, Image Compression and Poisson. We used twenty different combinations of C and W for each benchmark: $(C, W) = [0.01, 0.1, 0.5, 5, 50] \times [5, 50, 100, 500]$.

For each run, we measured the metrics described in Section 5.1 and used them to compute score function values. Due to space constraints, we focus on the resulting scores rather than individual metrics (we refer the curious reader to [9] for an in-depth analysis of the latter). We performed all tests on the `Xeon8` and `AMD48` systems (see Table 1). The reported numbers for `Xeon8` have been averaged over 30 runs, and the numbers for `AMD48` over 20 runs. The benchmarks are described in more detail in [2].

Acronym	Processor Type	Operating System	Processors
Xeon8	Intel Xeon X5460 3.16GHz	Debian 5.0	2 (×4 cores)
AMD48	AMD Opteron 6168 1.9GHz	Debian 5.0	4 (×12 cores)

Table 1. Specifications of the test systems used.

	static context				dynamic context			
	Xeon8		AMD48		Xeon8		AMD48	
	C	W	C	W	C	W	C	W
Sort	50.00	5	5.00	5	5.00	5	5.00	5
Bin Packing	0.01	5	0.10	5	5.00	500	5.00	500
Poisson	50.00	500	50.00	500	0.01	500	5.00	5
Image Compression	0.10	100	50.00	50	0.01	100	50.00	50

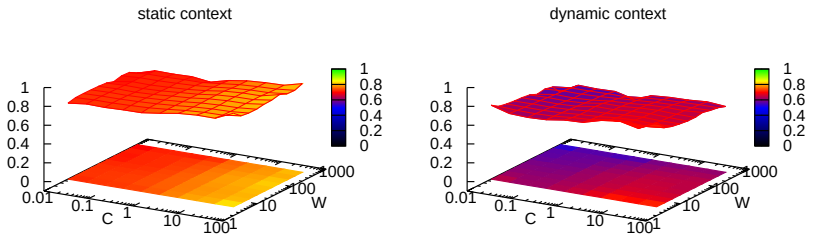
(a) Best performing values of the hyperparameters C and W over an empirical sample.

	static context		dynamic context	
	Xeon8	AMD48	Xeon8	AMD48
Sort	0.8921	0.8453	0.9039	0.9173
Bin Packing	0.8368	0.8470	0.9002	0.9137
Poisson	0.8002	0.8039	0.8792	0.6285
Image Compression	0.9538	0.9897	0.9403	0.9778

(b) Scores of the best performing hyperparameters.

Fig. 1. Best performing hyperparameters and associated score function values under static and dynamic autotuning scenarios.

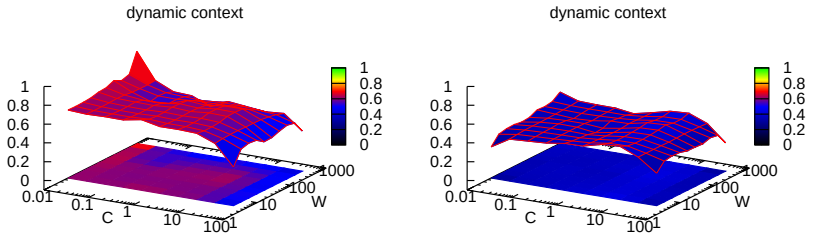
Figures 2 and 3 show select scores as a function of C and W on the `Xeon8` amd `AMD48` systems for benchmarks in both static and dynamic scenarios. All benchmarks except Image Compression show moderate to high sensitivity to hyperparameter values, with Bin Packing performance ranging from as low as 0.1028 at $(C, W) = (0.01, 5)$ to as high as 0.9002 at $(C, W) = (5, 500)$ in the dynamic scenario on the `Xeon8`. On average, the dynamic context was harder to autotune with a mean score of 0.6181 as opposed to static system's 0.6919 (Figure 4). This result confirms the intuition that maintaining a high average throughput while minimizing convergence time is generally more difficult than finding a very high-throughput candidate after a longer autotuning process.



(a) Sort on `Xeon8`

Fig. 2. Scores for the Sort benchmark as a function of C and W. The colored rectangle is a plane projection of the 3D surface and is shown for clarity.

The optimal hyperparameter values for each benchmark ranged considerably and depended on both the scenario and the architecture (Table 1). Sort tended to perform best with a moderate C and a low W, underlining the importance of exploration in the autotuning process of this benchmark. Bin Packing in the static context favored a balance between exploration and exploitation of a small number of recently tried operators. In the dynamic context Bin Packing performed best with much longer history windows (optimal $W = 500$) and with only a moderate exploration term $C = 5$. This is expected as Bin Packing in the dynamic context is comparatively difficult to autotune and hence benefits from a long history of operator performance. Poisson was another "difficult" benchmark, and as a result performed better with long histories ($W = 500$ for almost all architecures and contexts). In the static scenario it performed best with a high $C = 50$, confirming the authors' intuition that exploration is favorable if we are given more time to converge. In the dynamic context exploration was favored less (optimal $C = 0.01$ for the `Xeon8` and $C = 5$ for the `AMD48`). In the case of Image Compression, many hyperparameters performed close to optimum suggesting that it is an easy benchmark to tune. Medium W were preferred across architectures and scenarios, with $W = 100$ and $W = 50$ for the static and dynamic contexts, respectively. Image Compression on `AMD48` favored a higher $C = 50$ for both scenarios, as opposed to the low $C = 0.1$ and $C = 0.01$ for the static and dynamic contexts on the `Xeon8`. This result suggests exploitation of a limited number of well-performing operators on the `Xeon8`, as opposed to a more explorative behavior on the `AMD48`. We suspect this is due to a much higher parallelism of the `AMD48` architecture, where as parallelism increases different operators become effective.



(a) Poisson on `Xeon8` (left) and `AMD48` (right)

Fig. 3. Measured scores for the Poisson benchmark on each architecture.

6.1 Hyperparameter Robustness

Our results demonstrate that autotuning performance can vary significantly depending on the selection of hyperparameter values. However, in a real-world setting the user cannot afford to run expensive experiments to determine which values work best for their particular program and architecture. For this reason,

	static context		dynamic context	
	Xeon8	**AMD48**	**Xeon8**	**AMD48**
Sort	95.71%	100%	74.16%	61.12%
Bin Packing	85.61%	94.72%	67.42%	88.74%
Poisson	70.64%	71.09%	90.77%	96.07%
Image Compression	92.44%	96.35%	89.92%	91.42%

Table 2. Benchmark scores for the globally optimal values of hyperparameters normalized with respect to the best score for the given benchmark and scenario. The optimal hyperparameters were $C = 5$, $W = 5$ for the static context, and $C = 5$, $W = 100$ for the dynamic context. Mean normalized scores were 88.32% and 82.45% for the static and dynamic contexts, respectively.

we performed an empirical investigation whether there exists a single assignment of C and W that works well across programs and architectures.

We used the score functions from Section 5.1 to find hyperparameters that maximized the mean score on all the benchmarks. We found that the hyperparameters $(C, W) = (5, 5)$ for the static context and $(C, W) = (5, 100)$ for the dynamic context maximized this score. The results are shown in Table 2. For the sake of illustration, we normalized each score with respect to the optimum for the given benchmark and scenario (Table 1(b)).

Despite fixing hyperparameter values across benchmarks, we measured a mean normalized score of 88.32% for the static and 82.45% for the dynamic context, which means that we only sacrificed less than 20% of the performance by not tuning hyperparameters on a per-benchmark and per-architecture basis. This result shows that the hyperparameters we found are likely to generalize to other benchmarks, thus providing sensible defaults and removing the need to optimize them on a per-program basis. They also align with our results for individual benchmarks (Figure 1), where we found that exploration (moderate to high C, low W) is beneficial if we can afford the extra convergence time (static context), whereas exploitation (low to moderate C, high W) is preferred if average throughput and low convergence time are of interest (dynamic context).

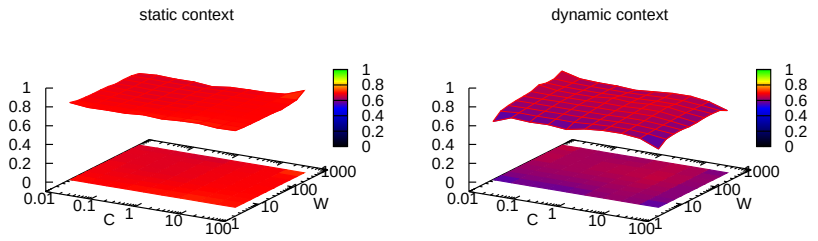


Fig. 4. Scores for the static and dynamic scenarios averaged over the Sort, Bin Packing, Poisson and Image Compression benchmarks and the `Xeon8` and `AMD48` architectures. The mean scores across all benchmarks, architectures and hyperparameter values were 0.6919 for the static and 0.6181 for the dynamic contexts.

7 Conclusions

We performed a detailed experimental investigation of hyperparameter effect on the performance of the PetaBricks autotuner, a real-world online evolutionary algorithm that uses adaptive operator selection. We evaluated four benchmarks with respect to three metrics which we combined into a performance indicator called the score function, and demonstrated that optimal hyperparameter values differ significantly between benchmarks. We also showed how two possible autotuning scenarios can affect the optimal hyperparameter values. We further demonstrated that a single choice of hyperparameters across many benchmarks is possible, with only a small performance degradation. Such a choice provides sensible defaults for autotuning, removing the need for the user to tune hyperparameters per-program, and thus making our approach feasible in a real-world setting.

References

1. J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.
2. J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization*, Chamonix, France, Apr 2011.
3. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47:235–256, May 2002.
4. A. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 3(2):124 –141, July 1999.
5. A. Fialho. *Adaptive Operator Selection for Optimization.* PhD thesis, Université Paris-Sud XI, Orsay, France, December 2010.
6. A. Fialho, L. Da Costa, M. Schoenauer, and M. Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence – Special Issue on Learning and Intelligent Optimization*, 2010.
7. A. Fialho, R. Ros, M. Schoenauer, and M. Sebag. Comparison-based adaptive strategy selection with bandits in differential evolution. In R. S. et al., editor, *PPSN'10: Proc. 11th International Conference on Parallel Problem Solving from Nature*, volume 6238 of *LNCS*, pages 194–203. Springer, September 2010.
8. J. Maturana, A. Fialho, F. Saubion, M. Schoenauer, and M. Sebag. Extreme compass and dynamic multi-armed bandits for adaptive operator selection. In *CEC'09: Proc. IEEE International Conference on Evolutionary Computation*, pages 365–372. IEEE Press, May 2009.
9. M. Pacula. Evolutionary algorithms for compiler-enabled program autotuning. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2011.
10. K. Price, R. M. Storn, and J. A. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization (Natural Computing Series).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.