Online-Autotuning in the Presence of Algorithmic Choice

Philip Pfaffe, Martin Tillmann, Sigmar Walter, Walter F. Tichy
Karlsruhe Institute of Technology (KIT)
Institute for Program Structures and Data Organization
Karlsruhe, Germany
philip.pfaffe,martin.tillmann@kit.edu

***Abstract*—In this paper we explore the problem of autotuning the choice of algorithm. For a given task, there may be multiple algorithms available, each of which may contain its own set of tunable parameters and may provide optimal performance under different sets of inputs.**

Algorithmic choice is a type of tuning parameter which has not been well studied in the history of autotuning. To close this gap, we examine established autotuning techniques with regard to their ability of handling these parameters. We discuss the inadequacy of the state-of-the-art autotuning toolbox in manipulating algorithmic choice parameters and introduce four strategies to tackle this task.

We evaluate our strategies in two case studies of online-autotuning scenarios, both with and without additional, numeric tuning parameters. The strategies are able to determine the optimal algorithm, and can even interoperate with the autotuning of the additional parameters.

***Keywords*-autotuning; online-autotuning; algorithmic choice; nominal parameters;**

I. Introduction

The choice of the optimal algorithm is a key problem in designing high performance software. This choice, however, can rarely be made in a general, a priori way to achieve maximum performance. The variations in data sizes, data types, and even system properties such as processor speed and parallelism make this impossible. Worse even, this variation can occur during application runtime.

This problem can be addressed with the help of autotuning. Autotuning is a tool designed to find optimal configurations for parameterized applications for a given value function. By systematically exploring the search space of possible configurations using some search strategy, the best configuration is iteratively found (or approximated). This method can be applied offline, e.g. as part of the installation procedure, to optimize the application for the concrete target machine. Alternatively, autotuning can be performed online, during application runtime, to also account for variations in input data and size. While in an offline scenario it is perfectly feasible to exhaustively try every possible configuration, online tuning exhibits much stricter real time constraints. Thus, a large body of fast (and approximative) search techniques has been proposed throughout the years in the autotuning literature, to speed up both offline and online search.

Multiple previous projects exist that have applied autotuning to approach the problem of algorithmic choice. However, these limit themselves to offline tuning, often enhanced with machine learning to provide some form of online adaptivity. In this paper, we will reexamine the fundamental properties of tuning algorithmic choice, and we will propose a set of methods designed for its optimization in online scenarios. Moreover, we will demonstrate how to combine the tuning of the parameter governing algorithm selection with autotuning of parameters exposed by the individual algorithms themselves. In two case studies, we evaluate our methods on two real-world applications, which offer multiple alternative algorithms in performance critical stages of their processing: We look at parallel string matching and raytracing, which have applications in multiple fields of research and industry. In these case studies we show that our methods are able to find the optimal algorithm, and are interoperable with autotuning the performance-relevant parameters of the individual algorithms.

The remainder of this paper is organized as follows. In the next section we will define the autotuning problem, and review common optimization techniques used in online-autotuning. We classify tuning parameters and algorithm selection parameters, and discuss the insufficiencies of established techniques in this parameter class. In Section III we will introduce a new set of optimization methods specifically for the class of nominal parameters and algorithmic choice. We present the evaluation of our methods in Section IV. The relevant related work is discussed in Section V. Section VI concludes this paper by summarizing our findings and discussing future work.

II. Background

In the following we give a brief overview of performance autotuning. We define the autotuning problem and review the most common optimization techniques. Based on a classification of tunable parameters we then analyze the problem of tuning algorithmic choice, and analyze the ability of the optimization techniques to handle algorithmic choice.

A. Autotuning

Autotuning is the process of automatically optimizing the performance of an application by iteratively configurable application parameters, searching for the optimal configuration. Application performance is interpreted with respect to application runtime, but may also refer to different metrics, such as energy consumption. In an offline tuning scenario, the autotuner usually chooses a configuration of the tunable parameters, then executes the application while observing its performance, and repeats this process until some termination criterion is met. In an online tuning setup, this *tuning loop* is found further inside the application. As a consequence, not every application lends itself to online tuning: The application must implement some operation which is executed repeatedly and is central to the applicaton's performance. This operation must further allow for the parameter configuration to change between tuning loop iterations. Its performance should only depend on the current configuration, as approximative search techniques tend to be vulnerable to measurement noise. In this work we focus on online autotuning, although the technique we develop here is applicable to offline tuning as well. The difference between online and offline tuning is mostly a technical one. Because an online tuning operates at application runtime, it is subject to stricter real time constraints.

More formally, both offline and online autotuning can be defined as the process of finding the minimum of a given *measurment function* $m_K : T \rightarrow \mathbb{R}$ defined as

$$C_{opt,K} = \underset{C \in T}{\arg\min}\, m_K(C).$$

For a given *context* $K = (K_A, K_S)$, describing the application A running on the system S, the measurement function m_K maps application *configurations* $C \in T$ onto application specific measurement values. In practice, m_K often measures the application runtime or the systems energy consumption. In this work we will assume m_K to be a measurement of time. For the sake of simplicity, the context is usually assumed to be constant during the tuning process. To ease readability we will therefore omit the context within following definitions throughout the remainder of this paper, implying that all conclusions we draw apply only within a fixed, but arbitrary context.

The configurations C are points in a J-dimensional *search space* T, which is composed from a finite set of *tuning parameters* τ_j:

$$T = \tau_0 \times \tau_1 \times \ldots \times \tau_J.$$

Often, the τ_j are implemented as closed integer intervals.

Applying Steven's typology[1], τ_j may be classified into one of four categories: Nominal, Ordinal, Interval, or Ratio Parameters. Their properties are summarized in table I. Every class is characterized by a distinguishing property, and subsumes the properties of all previous classes.

Minimizing m is a global optimization problem. Exhaustive exploration of the search space is generally too expensive in both online and offline scenarios. Therefore, research and real world applications rely on efficient search strategies to approximate the global optimum. In the following we give a brief overview of several techniques that are frequently applied in autotuning literature and by practitioners.

1) Hill Climbing: In every iteration, the hill climbing method evaluates the neighbors of a current solution candidate, and greedily moves towards the neighbor with the highest value. The method converges once there is no better neighbor.

2) Downhill Simplex: Also named the Nelder-Mead-Algorithm[2], this method maintains the nodes of a simplex in the search space, and moves and contracts this simplex towards an extremum (possibly local), using a small state-machine of simplex transitions. This method is frequently used in practice because it often shows very quick convergence.

3) Particle Swarm: The particle swarm optimization[3] searches an optimum by maintaining a set of candidate solutions. Candidates are iteratively updated by an individual local "velocity".

4) Genetic Algorithms: Genetic algorithms[4] are motivated by biological evolution. A new configuration is obtained either through "mutation", by randomly modifying one or more parameters, or through "crossover", by interleaving two old configurations at a random crossover point.

5) Differential Evolution: Differential evolution[5] operates on a set of candidate solutions referred to as agents. An agent is updated based on three randomly selected agents. Every dimension of the agent is probabilistically updated based on the differences of the three selected agents in this dimension.

6) Simulated Annealing: The simulated annealing method[6] is motivated by the physical process of a cooling material. In its essence, the method is identical to hill climbing. However, in every hill climbing step, there is a predefined chance of taking a step in a non-optimal direction, thus reducing the probability of arriving in a local minimum.

7) Exhaustive & Random Search: Although these techniques are neither efficient nor sophisticated we see the need to also mention exhaustive and random search. The semantics of these are self explanatory: try every possible configuration systematically, or roll the dice in every iteration, respectively. Exhaustive search is often applied in offline tuning scenarios, and if the search space is small enough that the search can be completed in a comfortable time frame. Random search is rarely used in practice.

B. Autotuning Algorithmic Choice

Algorithmic choice, i.e. selecting an algorithm for a given problem from a set of alternatives, represents an instance of a nominal parameter. Algorithms, if they take the same inputs

[1]Note that the percentage itself is an interval value, however the actual physical buffer size it represents is not.

Table I
PARAMETER CLASSES

Class	Distinguishing Property	Example
Nominal	Labels	Choice of algorithm
Ordinal	Order	Choice of buffer sizes from a set `small`, `medium`, `large`
Interval	Distance	Percentage of a maximum buffer size[1]
Ratio	Natural Zero, Equality of Ratios	Number of threads

and produce the same outputs, can not be ordered, do not offer a notion of distance and do not have a natural zero point. To apply autotuning to algorithmic choice therefore requires a tuning method that is able to handle nominal parameters. The obvious first choice is of course exhaustive search. This search technique is perfectly valid if algorithmic choice is the only parameter that is being optimized, or, more generally, if the search space is comprised entirely of nominal tuning parameters. Then, trying one configuration gives us no information about any other possible configuration, and picking every possible value once is optimal. If, on the other hand, the search space contains a mixed set of parameter classes, for instance when algorithms expose non-nominal tunable parameters themselves, exhaustive search becomes less adequate. It is necessary to minimize the time spent searching because we wish to optimize total application performance, which in an online tuning scenario includes the search. Although exhaustive search is guaranteed to eventually select the best configuration, it will also always select the worst configuration. The autotuning happens at runtime, therefore the costs of selected configurations has to be amortized. Thus, a search technique is required that focuses on the more promising candidates, based on the information that can be derived from the non-nominal tuning parameters.

Of the algorithms introduced in the previous section, only genetic algorithms define a meaningful way of manipulating the nominal parameter type. The Hill Climbing method and by extension Simulated Annealing require a notion of *neighborhood*. Differential Evolution operates on the *difference* of configuration. Both Nelder-Mead and Particle Swarm operate on a measure of *direction* and *distance*. Genetic algorithms in turn do not require any of these measures, which enables them to operate on nominal parameter spaces. They are, however, applicable to the tuning of algorithmic choice only in a very limited fashion. Selecting the algorithm is only a single nominal parameter. Using either of the mutation strategies discussed above thus turns the genetic algorithm into a random search. On the other hand, if we consider tunable parameters exposed by the individual algorithms themselves, genetic algorithms can be applicable, however at the cost of losing the performance benefits offered by alternatives such as the Nelder-Mead method in these parameter spaces.

Existing approaches for algorithmic choice (see Chapter V) find several ways around this issue. PetaBricks[7] converts the nominal parameter into a ratio parameter, by linking algorithms to input sizes. The Nitro[8] framework operates similarly, based on user-defined features extracted from input data.

III. TUNING ALGORITHMIC CHOICE

The algorithmic choice tuning problem can be modeled as a two-phase tuning problem. Given a set of algorithms $\mathcal{A}$, the tuning problem becomes

$$C_{opt} = \underset{A \in \mathcal{A}, C \in T_A}{\arg\min}\; m_A(C),$$

for the updated measurement function $m_{A,K} : T_A \to \mathbb{R}$. As distinct algorithms do not necessarily share tuning parameters, parameter spaces are modeled as one tuning parameter space per algorithm, T_A, which need not necessarily be disjoint (although in practice they usually are). Now, the (globally) optimal configuration C_{opt} contains the optimal algorithm, as well as the configuration of that algorithm's tuning parameters.

The two-phase formulation of the tuning problem enables us to tackle the optimization for each algorithm individually, by first determining

$$C_{opt,A} = \underset{C \in T_A}{\arg\min}\, m_A(C)$$

for every $A \in \mathcal{A}$. In the second phase, the global optimum is determined as

$$C_{opt} = \underset{A \in \mathcal{A}, C = Copt,A}{\arg\min}\; m_A(C).$$

Numerous well-studied approximative solutions for the first phase exists. However, since algorithmic choice parameters are nominative in nature, these solutions are not applicable to the second phase. We hence devise four strategies to address the second phase, which we further discuss below.

To approximate the optimal configuration of the algorithm choice and the respective tuning parameters, we iteratively apply both phases to the tuning problem in reverse order. In tuning iteration i we first select an algorithm A using one of the phase-two strategies. We then determine a tuning parameter configuration C_i for A using a phase-one strategy. In our case studies we rely on the Nelder-Mead downhill simplex method in this step. Observing the runtime performance of A with the configuration C, we obtain a runtime sample $m_{A,i} = m_A(C_i)$. We repeat this process indefinitely or until a user-defined termination criterion is met.

In the remainder of this section we will discuss four probabilistic strategies which we use to select an algorithm in every iteration.

A. The ϵ-Greedy Strategy

The ϵ-Greedy strategy is a parameterized probabilistic method which selects the currently best performing algorithm with a probability of $1-\epsilon$. Otherwise, an algorithm is chosen at random with uniform probability. In this strategy, ϵ is a configurable parameter which enables direct control of the explorative behavior of this method. We use ϵ-values of 5%, 10%, and 20% in our case studies.

This strategy is probably most well known for its application in the field of Reinforcement Learning as an action selection policy. There, an agent selects an action for a given state by either *exploiting* knowledge about the best know action for this state, or by *exploring* different actions. Another frequently applied alternative to the ϵ-Greedy policy is a soft-max[9] policy, which chooses an action during exploration according to a given probability, most commonly using a Gibbs distribution. If some actions produce significantly worse results than others, this policy helps to avoid those actions. In our application however, we explicitely do not want to avoid bad algorithms (which correspond to the actions in a Reinforcement Learning framework), to allow them to improve over time due to the second-phase tuning.

B. The Gradient Weighted Strategy

The Gradient Weighted strategy is a probabilistic method which chooses an algorithm $A \in \mathcal{A}$ with probability proportional to a weight w_A in every iteration, based on the gradient observed in the performance samples of the latest iteration window $[i_0, i_1]$ of A. We define w_A as

$$w_A = \begin{cases} G_A + 2 & \text{if } G_A \geq -1 \\ -\frac{1}{G_A} \end{cases}$$

with $G_A = \frac{\frac{1}{m_{A,i_1}} - \frac{1}{m_{A,i_0}}}{i_1 - i_0}$. Note that in this definition, we interpret "performance" inversely to the measured samples $m_{A,i}$ to support intuition: an algorithm is considered "more performant" the less time it consumes. Further note that in our definition above the values of w_A is always positive. Thus, we never exclude an algorithm from the selection process, and the selection probability of algorithm A is $P_A = \frac{w_A}{\sum_{A' \in \mathcal{A}} w_{A'}} > 0$.

In our case studies, we used an iteration window of 16.

C. The Optimum Weighted Strategy

The Optimum Weighted strategy is a probabilistic method which chooses an algorithm $A \in \mathcal{A}$ with a probability relative to its current optimal performance $w_A = \max_i \frac{1}{m_{A,i}}$. Again, the weight w_A is strictly positive, and the selection probability of algorithm A is $P_A = \frac{w_A}{\sum_{A' \in \mathcal{A}} w_{A'}} > 0$.

D. The Sliding Window Area-Under-The-Curve Strategy

The Sliding Window AUC strategy is again a probabilistic method, which assigns a weight w_A based on the area under the algorithm's performance curve within a sliding iteration window $[i_0, i_1]$ of A. This strategy is motivated by the AUC Bandit meta heuristic described in the OpenTuner[10] article. We define w_A as

$$w_A = \frac{\sum_{i=i_0}^{i_1} \frac{1}{m_{A,i}}}{i_1 - i_0}$$

and the selection probability as before as $P_A = \frac{w_A}{\sum_{A' \in \mathcal{A}} w_{A'}} > 0$.

In our case studies, we used a window size of 16.

E. Genetic Algorithms

While the nature of genetic algorithms enables configuration approximation in nominal parameter spaces, they do not significantly contribute to solving the problem discussed in this paper. Because there is only a single parameter that we manipulate, the common genetic mutation strategies are not applicable. Nonetheless, alternate mutation strategies are imaginable. However, again because there is only a single parameter, these strategies decay to one of the above.

Table II
SPECIFICATIONS OF THE BENCHMARK SYSTEM

Processor	Intel Xeon E5-1620v2
Speed	3.70GHz
Threads	8
RAM	64GB

IV. Evaluation

We assess the ability of our four nominal parameter tuning strategies to effectively manipulate algorithmic choice parameters using two case studies.

In the first case study, we will look at parallel string matching. The basis for this is our 2016 paper[11], in which we presented parallel versions of several state-of-the-art string matching algorithms. The algorithms themselves do not expose any tunable parameters. In this study we will observe the basic behavior of the search strategies.

The second case study covers a tunable raytracing application. This application stems from the publication by Tillmann et al.[12], and makes use of four different algorithms for constructing a data structure fundamental to raytracing performance. These algorithms expose multiple tunable parameters. In this study we evaluate the effect of the combination of tuning the choice of algorithm with tuning the individual algorithms themselves.

A. Case Study 1: Parallel String Matching

String matching is a frequently employed tool with a wide array of applications. We investigated parallel versions of seven state-of-the-art string matching algorithms and evaluated their performance on multiple text corpora[11], including the text of the English King James Bible and the sequence of the human genome. We evaluate an online scenario: the query pattern and text corpora are supplied

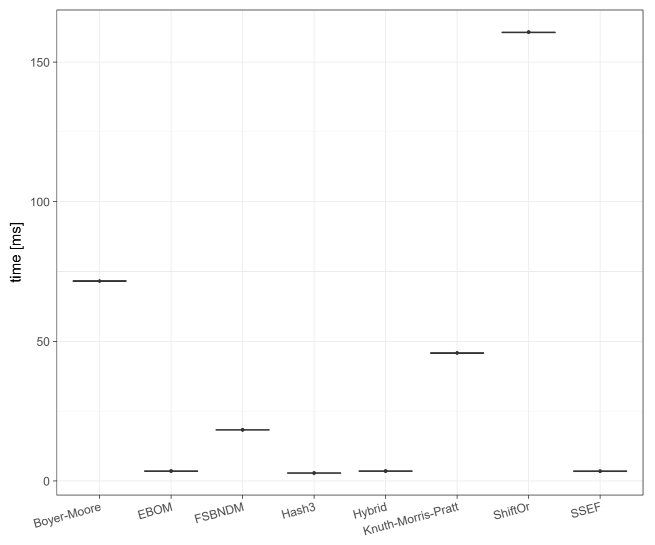

Figure 1. String Matching: Performance of the parallel string matching algorithms

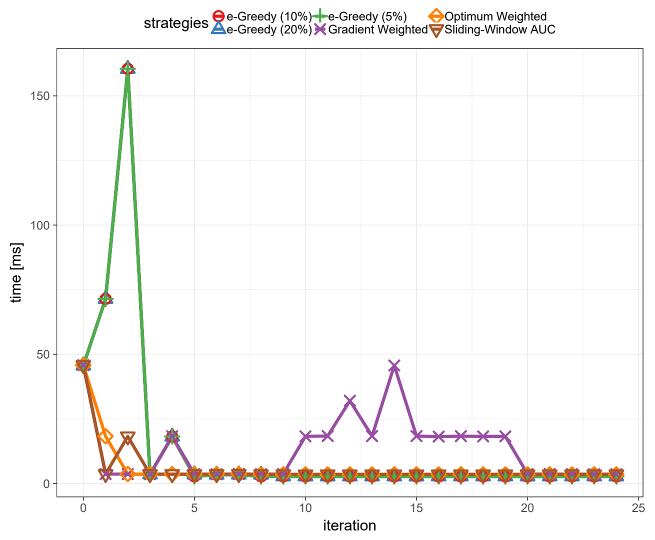

Figure 2. String Matching: Median performance in individual iterations of all strategies

at program invocation. Any precomputation is part of the algorithms runtime.

The seven string matching algorithms are `Boyer-Moore`, `EBOM`, `FSBNDM`, `Hash3`, Knuth-Morris-Pratt (`KMP`), `ShiftOr` and `SSEF`[2]. Additionally we implemented a heuristic-based string matcher, labeled `Hybrid`, that chooses one of the seven algorithms based on the pattern length. The algorithms all follow the same two phase pattern: first a precomputation is performed on the pattern. Then a skip-ahead heuristic is iteratively evaluated on the text to discard unfeasible text chunks, only checking the remaining possible matches. The parallelization of the algorithms is based around partitioning the input text. In all algorithms, each partition is processed by one thread. Where applicable, several of the algorithm implementations make use of bit parallelism and SSE intrinsics in the text search.

In this evaluation, we use all of the seven string matching implementations, searching for the query phrase "the spirit to a great and high mountain" within the English text of the Bible. The benchmark system is an Intel Xeon E5-1620v2 machine with 64GB of RAM, see table II for further details. We tune the string matching application for 200 iterations, each iteration repeating the search for query phrase. The length of this tuning loop is chosen to ensure tuning convergence. This experiment is repeated 100 times for stability. Figure 1 shows a boxplot of the performance results for the individual string matching algorithms without tuning on the bible benchmark. We see that four algorithms, namely `SSEF`, `EBOM`, `Hash3`, and `Hybrid`, yield the best performance.For a more detailed explanation of the individual algorithms we refer the interested reader to the source paper[11]. Since the boxes appear only as thin lines, these results are very stable.

If we now enable tuning of the algorithm selection, we observe the performance profile shown in figure 2. Here, we show the median time consumed in every iteration over the 100 experiments. Iterations are capped at 25 as all plots are converged to a stable value. The first thing we observe here is the effect of initialization. The ϵ-Greedy variants initialize by trying every inidividual algorithm exactly once in deterministic order, although this is still subject to the ϵ-randomness. This order is clearly visible in the first seven samples of the ϵ-Greedy curves. The remaining strategies do not treat initialization in a special way, except that they start with a deterministic configuration. The progression of the Gradient Weighted curve is another point of interest, which we discuss below in greater detail.

It is further noteworthy, that the ϵ-Greedy variants appear strictly in unison in this plot. This is firstly due to the fact that we're showing median values here, and secondly, because the exploration factor is much smaller than 50%, there is a high likelihood that the median is always the current best value.

For completeness we additionally show the mean performance for all iterations in figure 3. In the curves of the ϵ-Greedy variants we can see the effect of randomness in the initialization period in the disparity between the curves. The Gradient Weighted curve shows some unexpected results, however. Because there are no tunable parameters in the string matching implementations, we expect the performance measurements to be very much the same in every iteration, which should result in a gradient of 0. This in turn would result in the Gradient Weighted curve to behave like a

[2]The implementations are available at https://code.ipd.kit.edu/pmp/pgerp

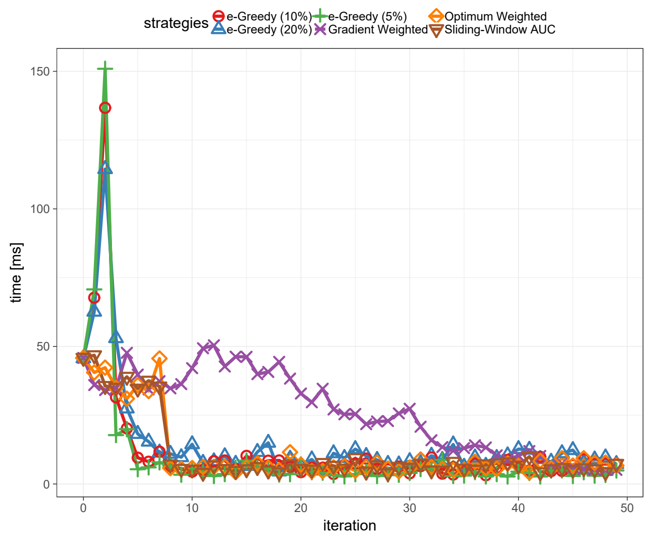

Figure 3. String Matching: Mean performance in individual iterations of all strategies

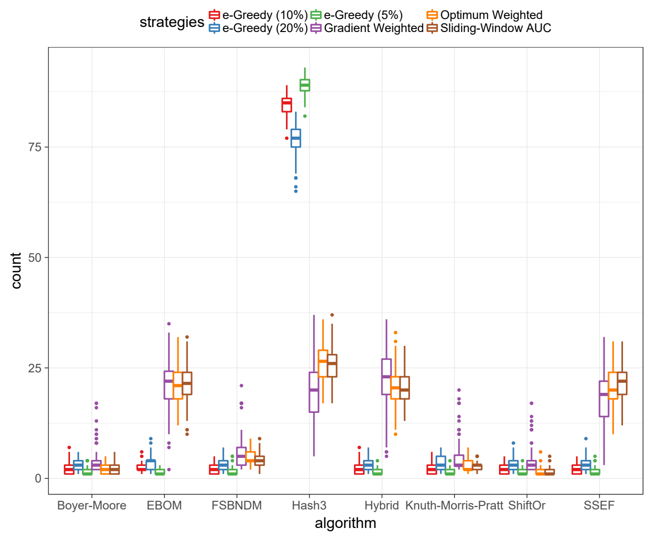

Figure 4. String Matching: Frequency of all algorithms being chosen by the strategies

random selection, with a relatively stable average equal to the runtime average over the untuned string matching performance results. In our data, however, we observe a converging trend as is visible in figure 3. We see the cause of this in the measurement noise: Although figure 1 suggests that the overall noise is very low, the standard deviations of `Boyer-Moore`, `KMP` and `ShiftOr` are an order of magnitude larger compared to the remaining algorithms (0.2 compared to 0.06).

Figure 4 shows the accumulated histogram of the algorithm choice for all strategies. The frequencies are shown as a boxplot over the 100 experiment repetitions. The plot shows clearly how all the Greedy strategies prefer the `Hash3`-algorithm, whereas Gradient Weighted, Optimum Weighted, Sliding-Window AUC also give consideration to `EBOM`, `Hybrid`, and `SSEF` with almost equal frequency. Although there is a slight difference visible in the plot, this difference is insignificant.

B. *Case Study 2: Raytracing*

In our second case study, we look at a raytracing application. The raytracing application renders a single static scene for 100 frames using a two stage rendering pipeline. In the first stage, a lookup datastructure, called an SAH kD-tree, is constructed to accelerate ray/primitive intersection queries. In the second stage, rays are cast from the camera into the scene and tested for intersection with the geometric primitives of the objects of the scene. If a primitive is hit, a second ray is cast toward the light sources to test for ambient occlusion. The version of the application used in this case study is identical to the version used by Tillmann et al.[12], and offers four different, heuristical algorithms, named `Inplace`, `Lazy`, `Nested`, `Wald-Havran`, to construct the acceleration data structure in the first stage, and a simple raycasting implementation in the second. The algorithms build the kD-tree over the input scene in parallel using OpenMP. They differ in the way they map geometric primitives to threads, e.g. by mapping tree nodes to OpenMP Tasks as in the `Wald-Havran` variant, or by relying on data parallelism as in the `Inplace` algorithm. The parallelization depth as well as the parameters of the SAH heuristic are tunable parameters in all algorithms. The `Lazy` algorithm adds another parameter, controlling the eager construction cutoff. For a more in-depth discussion of the construction algorithms and their relevant tuning parameters, refer to Tillmann et al.

Here, we run the raytracing application for the Sibenik scene from the original paper, rendering it for 100 frames for a single experiment and repeat this 100 times. The tuning loop is the rendering loop. For each frame, a construction algorithm along with its new parameter configuration is selected by the online tuner. The length of the tuning loop is chosen to guarantee tuning convergence in every experiment. Figure 5 displays the tuning profile for the frame rendering iterations, averaged over all repetitions. This plot gives an impression of how the Nelder-Mead online-autotuner optimizes the construction performance over time. Most noteworthy is the leap we see right on the first tuning iteration. This is due to the fact that for all construction algorithms the tuner starts off with a hand-crafted configuration which Tillmann et al. created based on best practices of the relevant literature.

When we combine the Nelder-Mead online-autotuner for the algorithms' own tuning parameters with our strategies for algorithmic choice, we obtain a performance profile as shown in figure 6. Here we see the combined effect of tuning

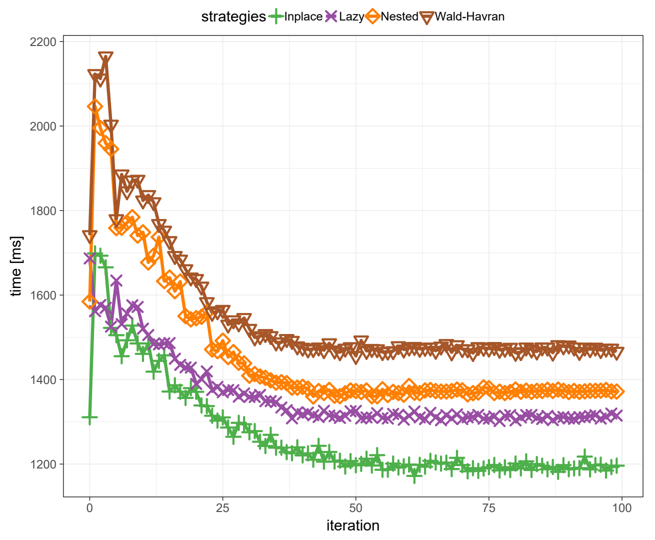

Figure 5. Raytracing: Tuning timeline of all four algorithms. The plot shows the average time taken in every iteration.

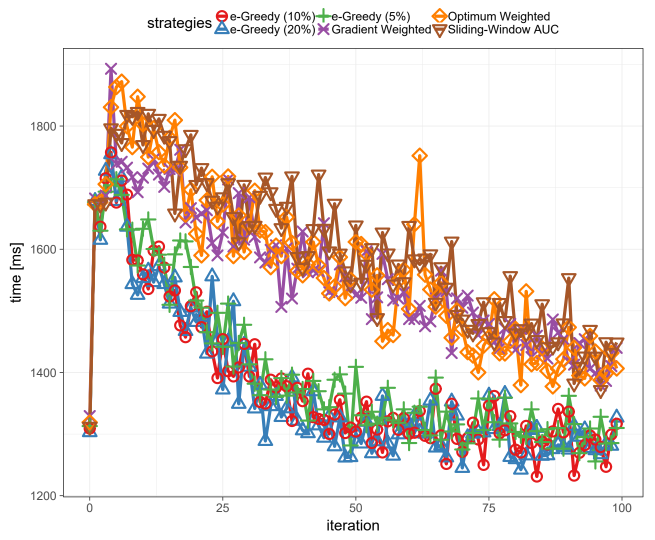

Figure 7. Raytracing: Mean performance in individual iterations of all strategies

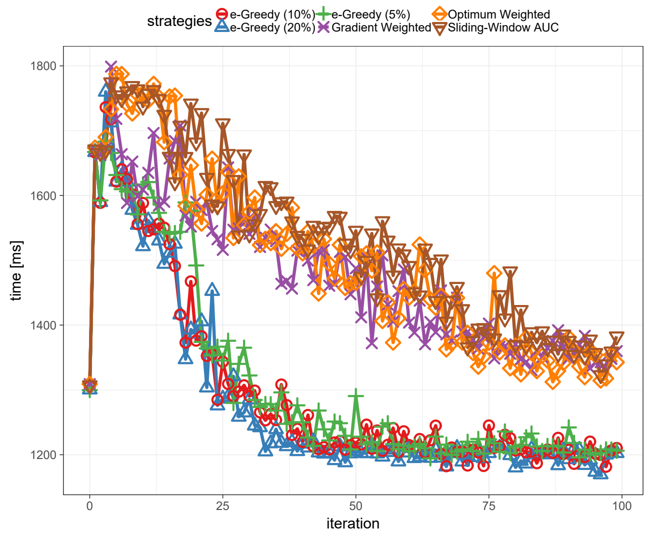

Figure 6. Raytracing: Median performance in individual iterations of all strategies

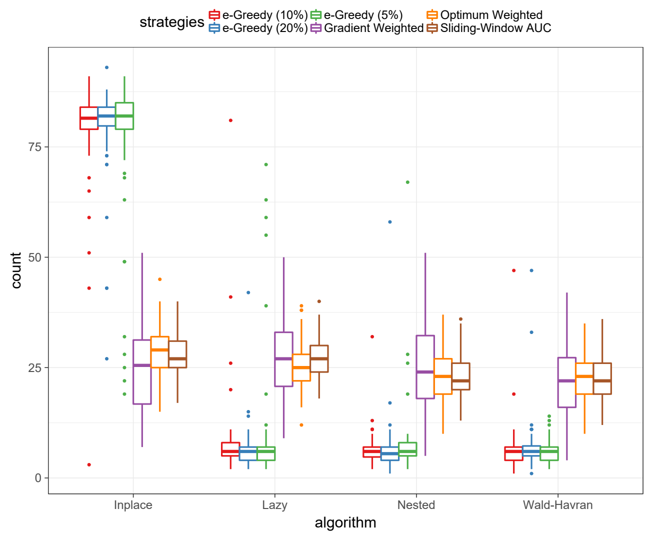

Figure 8. Raytracing: Frequency of all algorithms being chosen by the strategies

individual algorithms and selecting an algorithm in every frame. On the zeroth iteration, all strategies start off with the same algorithm. The ϵ-Greedy variants quickly identify the fastest of the construction algorithms and then converge on that. The remaining strategies switch back and forth between algorithms, and are able to achieve tuning progress on all algorithms more or less simultaneously. For completeness, figure 7 displays the averaged data for the same context, which shows the same properties as the median data. The large spike in the Optimum Weighted curve is caused by picking up a large number (of 4) experiment runs in which the `Nested` and `Wald-Havran` algorithms choose a particularly bad configuration, whose durations are off by a factor of 5. In our data, this occurs multiple times for the `Nested` and `Wald-Havran` runs, however scattered over different iterations in most cases.

Figure 8 again shows the accumulated histograms for all strategies and algorithms as a boxplot over the experiment runs. Again we see that the ϵ-Greedy variants concentrate on the overall fastest algorithm. The remaining strategies do not show a significant preference toward any single algorithm. For the Gradient Weighted method the reason can be seen in figure 5. The Gradient Weighted method prefers algorithms which gain significant performance improvements within a window of tuning iterations over those which do not. In the algorithms used here, however, the autotuning progression made by the Nelder-Mead autotuner is, on average, relatively similar. The Gradient Weighted method can thus not differentiate between the algorithms. In the remaining two methods, the failure to discriminate algorithms is caused by the way they calculate weights. The procedure is rather similar in both cases, as they make decisions based on the observed absolute performance within an iteration window. To calculate weights, Optimum Weighted relies on the maximum norm, whereas Sliding Window AUC calculates the average. Because the difference of the absolute performance of all algorithms is (on average) small, both strategies are unable to identify the fastest algorithm easily.

C. Discussion

In our case studies, we have seen that ϵ-Greedy is able to pick the best algorithm. This is the case whether the algorithms are subject to tuning themselves or not.

A threat to the validity of this conclusion is rooted in the tuning progression of the algorithms in the second case study. As we see in figure 5, the (average) performance improvement profile of the algorithms is strikingly similar. Thus, we are unable to predict how the ϵ-Greedy strategy will behave if the tuning profile contains a crossover point, however unlikely this may be in practice. In this situation, ϵ-Greedy might take very long to converge to the second algorithm with better post-tuning performance. We anticipate to be able to mitigate this drawback by combining the strategies we have presented here, in particular with the Gradient-Weighted method.

The Gradient-Weighted method as we presented it here is a special case, which we do not expect to be applicable in practice. With this method, once the tuning of all algorithms has converged, the algorithm selection will jump randomly between algorithms in the set, without regard to their individual absolute performance. We included it in this paper and our case studies as a possible means to mitigate drawbacks of the ϵ-Greedy strategy.

V. Related Work

The need for autotuning software was first identified in high peroformance and scientific computing. The field has since recieved extensive attention. Two representatives of autotuning in scientific computing are FFTW[13] and the ATLAS project[14]. FFTW provides an adaptive software architecture, which uses empirical offline tuning of FFT solvers for the current system. The ATLAS project offers linear algebra routines, automatically tuned for the current system by fully exploring the configuration space.

Several general purpose tuning frameworks have since been proposed, most prominent among them Active Harmony[15]. This framework is built for online tuning of applications in a distributed context. Application instances report performance metrics to a centralized tuning controller, which supports a variety of search techniques to generate individual application configurations.

Autotuning is not limited to optimizing application runtime, but is also able to operate in multi-objective scenarios. Jordan et al. have built Insieme[16], a compiler and runtime environment for multi-objective tuning of applications for runtime performance and energy efficiency. Insieme follows a hybrid approach, generating the multi-objective pareto front during compiletime, and switching between pareto-optimal configurations at runtime.

Although the history of autotuning dates back two decades, autotuning of nominal parameters has, to the best of our knowledge, not been studied explicitly in the relevant literature. There are, however, several works available that operate on special cases of at least partially nominal parameter spaces without identifying them as such. Instances of this can be found in several projects[17], [18], [19] which manipulate the sequence of compiler opimizations applied during compilation. Tuning of algorithmic choice can also be found in domain specific systems. SPIRAL[20] is a digital signal processing tuner that optimizes user specified transforms by generating a formula in a mathematical description language. Tuning is applied to optimize the formula before code is generated. The OpenTuner project[10] is dedicated to optimize another type of nominal parameter, and offers a meta-tuner which tries to find the optimal search technique for a given tuning problem. The meta-tuner search strategy is similar in nature to our Sliding Window AUC method. As OpenTuner provides a general purpose tuning framework, Ansel et al. recognize the existence of nominal parameters, but leave it to the user to define a meaningful manipulation method.

Furthermore, there are several projects that approach the problem of algorithmic choice. Most recently, there have been several publications around the work of Ansel et al. and the PetaBricks language and compiler[7], [21]. The PetaBricks language is a parallel programming language which offers algorithmic choice as a first class construct. Algorithms that solve the same problem on the same inputs are labeled as alternatives. Offline autotuning on the target system is then used to build a decision tree model for choosing the optimal algorithm at runtime based on input data characteristics[22]. A similar approach is taken by the Nitro framework[8]. Muralidharan et al. use machine learning to train a model over algorithm specific input data characteristics (e.g. denominating the sparsitiy of a matrix). ADAPT[23] and the work by Tiwari and Hollingsworth[24] yield two examples which offer compiler-based algorithmic choice. Using dynamic code generation and online-autotuning, compiler optimization variants are applied speculatively, and evaluated at application runtime.

VI. Conclusion

We have presented the problem of online-autotuning algorithmic choice. By classifying tuning parameters and reviewing common autotuning techniques, we identified the shortcomings of these techniques with regard to selecting the optimal algorithm. To mitigate these shortcomings we devised four alternative methods capable of handling nominal parameters, of which algorithmic choice is an instance. We evaluate our new techniques in two case studies, first examining them in isolation on a string matching application, then in collaboration with search-based autotuning of parameters of the individual algorithms in a raytracing scenario.

In our case studies, we demonstrate the ability of our techniques to find the optimal algorithm. We show that our ϵ-Greedy strategy is able to achieve fastest convergence both in the presence and absence of additional, non-nominal tuning parameters. The remaining strategies achieve convergence as well but at a slower rate.

In the future we will expand on this work by generalizing from the problem of algorithmic choice towards arbitrary nominal parameters. This requires combining the techniques presented here to achieve maximum convergence speed while defending against local extrema. Evaluating this will call for a new set of benchmarks, that combines nominal with non-nominal parameters.

Acknowledgment

This work is partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – TI 264/10-1.

References

[1] S. S. Stevens, *On the Theory of Scales of Measurement*. Bobbs-Merrill, College Division, 1946.

[2] J. A. Nelder and R. Mead, “A Simplex Method for Function Minimization,” *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965.

[3] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *IEEE International Conference on Neural Networks*, 1995.

[4] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[5] R. Storn and K. Price, “Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces,” *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by Simulated Annealing,” *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[7] 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*, 2009.

[8] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, “Nitro: A Framework for Adaptive Code Variant Tuning,” in *IEEE International Parallel and Distributed Processing Symposium*, 2014.

[9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 1st ed. Cambridge, Mass: A Bradford Book, Mar. 1998.

[10] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O’Reilly, and S. Amarasinghe, “OpenTuner: An extensible framework for program autotuning,” in *International Conference on Parallel Architectures and Compilation*, 2014.

[11] P. Pfaffe, M. Tillmann, S. Lutteropp, B. Scheirle, and K. Zerr, “Parallel String Matching,” *International Workshop on Multicore Software Engineering*, 2016.

[12] M. Tillmann, P. Pfaffe, C. Kaag, and W. F. Tichy, “Online-Autotuning of Parallel SAH kD-Trees,” in *IEEE International Parallel and Distributed Processing Symposium*, 2016.

[13] M. Frigo and S. G. Johnson, “FFTW: An adaptive software architecture for the FFT,” in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 1998.

[14] R. Clint Whaley, A. Petitet, and J. J. Dongarra, “Automated empirical optimizations of software and the ATLAS project,” *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001.

[15] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth, “Active Harmony: Towards Automated Performance Tuning,” in *ACM/IEEE Conference on Supercomputing*, 2002.

[16] H. Jordan, P. Thoman, J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, “A multi-objective autotuning framework for parallel codes,” in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[17] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, “Finding Effective Compilation Sequences,” in *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2004.

[18] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O’Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, “Using Machine Learning to Focus Iterative Optimization,” in *International Symposium on Code Generation and Optimization*, 2006.

[19] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan, “Predictive Modeling in a Polyhedral Optimization Space,” *International Journal of Parallel Programming*, vol. 41, no. 5, pp. 704–750, 2013.

[20] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, “SPIRAL: Code Generation for DSP Transforms,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[21] J. Ansel, “Autotuning programs with algorithmic choice,” Ph.D. dissertation, Massachusetts Institute of Technology, 2014.

[22] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O’Reilly, and S. Amarasinghe, “Autotuning Algorithmic Choice for Input Sensitivity,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[23] M. J. Voss and R. Eigemann, “High-level Adaptive Program Optimization with ADAPT,” in *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 2001.

[24] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, “A Scalable Auto-tuning Framework for Compiler Optimization,” in *IEEE International Parallel and Distributed Processing Symposium*, 2009.