Tuning Computer Systems with Structured Bayesian Optimization

Valentin Dalibard, Eiko Yoneki
Cambridge University Computer Laboratory
valentin.dalibard@cl.cam.ac.uk

Example Problem: Build a scheduler for a distributed system

- What heuristics to use?
- How to detect if a worker has failed?

Current Approach I: Manually tune the system

✓ Makes use of programmer's understanding
✓ Profiling tools to see runtime properties
✗ Needs a lot of programmer time
✗ Solution is specific to the environment and input

Current Approach II: Use a Black-Box Optimization Tool

✓ Requires little programmer time
✓ Can be re-run to adapt to new environments and input distributions
✓ Can surpass human expert-level tuning
✗ Can involve thousands of evaluations of the objective function
✗ Does not scale to large parameter spaces

Proposed Approach: Structured Bayesian Optimization

Bayesian Optimization
The optimization iteratively builds a probabilistic model of the objective function:

- Find a set of parameter values with high performance in the model
- Update the model to reflect this new measurement
- Evaluate the objective function at that point

Structured Bayesian Optimization
We add two constructs to let users expose the structure of their programs:

Construct I: User-Given Models
Users specify the model to be learned using our probabilistic programming framework.
- Faster convergence
- Can reason about parameter values in a complex parameter space
- Can leverage all measurements reflected in the model

Construct II: Sub-Experiments
Users can declare cheap sub-experiment that can be used to perform inference on the model.
- Can quickly discard poor parameter values

Proof of concept: Decision trees for `std::sort()`

- We extract the parameters of `std::sort()` in libstdc++
- Construct a decision tree parameter space which queries the array's length and sortedness to pick parameter values
- Can generate bespoke high performance decision trees for arbitrary input distributions and environments

Outcomes:
- We systematically outperform `std::sort`
- Our optimized implementations have the high performance on their own input distributions (white diagonal)
- No implementation performs well on all input distributions, showing the need for an auto-tuned implementation.

```cpp
count_unsorted(input.begin(),
               input.begin() + 150) > 10
sort(input, 13) sort(input, 7)
```
Consider a programmer implementing a scheduler for their distributed system. Although the overall structure is fixed, many parameters, such as the ones in the scheduling heuristics, allow for some implementation freedom. The values of those parameters do not affect correctness, but they do impact performance and hence they must be tuned to produce an efficient implementation.

Currently, two solutions are available. On one hand, the programmer could manually tune these parameters. They would do this by leveraging some profiling tools as well as their understandings of the tradeoffs and bottlenecks involved. On top of requiring a fair amount of programmer’s effort, this approach has the disadvantage of not being robust to changes in the environment. If the network bandwidth decreases, or the input distribution changes, the tuning process will have to be executed again to account for this new context. This issue is especially relevant to computer systems which are designed to abstract an environment and hence should be performed in a wide range of settings.

On the other hand, the programmer could use a black-box optimization tool. To do this, they would expose the parameters of their scheduler to an optimizer and provide as objective function some performance metric of their system, such as the average job runtime. Recent work has shown that, when tuning program parameters, automatic optimization tools could surpass human expert-level optimizations. Furthermore, using this method different environments can simply be tackled by running the optimization again. However, black-box optimization tool make little assumptions about the problem at hand, and hence can be inefficient. Practically, this means they may need up to thousands of evaluation of the objective function, and will have difficulty scaling beyond dozens of parameters.

In this work we extend Bayesian optimization, often used for black-box optimization, in an attempt to bridge these two approaches. We present Structured Bayesian Optimization (SBO), a new framework which lets users expose the structure of their programs to guide the optimization towards high performance parameter values.

Traditional Bayesian optimization works by incrementally building a probabilistic model of the objective function. Each iteration, a point with high performance in the model is picked and evaluated. The model is then updated with this new observation.

SBO extends Bayesian optimization with two new constructs. First, rather than learning a general probabilistic model, such as a neural network or a Gaussian process, we allow users to specify structured models via our probabilistic programming framework. Probabilistic programming is a recent tool from the Machine Learning community which generalizes graphical models and makes the construction of structured probabilistic models intuitive.

This approach allows users to construct finer grained models which capture their knowledge about the behavior of the program, essentially shifting the uncertainty from the impact of the parameters themselves, to the the tradeoffs between high level features such as the locality of the computation or its effect on performance. These models can decouple data properties, such as the input distribution, from environment properties, and leverage all runtime measurements to perform inference.

Second, we let users declare sub-experiment that can be used to perform inference on the model at a low cost. These can be used to quickly discard poor parameter values and focus the use of the expensive utility function on the region of interest.

As a proof of concept, we extract the parameters of libstdc++’s std::sort() implementation. We find the optimal parameter values are highly dependent on the input’s length and how sorted it already is. Using our framework we build a decision tree parameter space which, given an input distribution and environment properties, generates a bespoke high performance dynamic dispatcher, trading-off query costs and quality of dispatch. We generate optimized implementations for a range of input distributions, and evaluate their performance.

Our results show three important outcomes. First, our optimized sort implementations consistently outperform std::sort(). This is because the std::sort() implementation was tuned to perform well on purely random inputs and is therefore sub-optimal in other settings. Second, for each input distribution, the associated implementation always performs well. This confirms that we are able to generate high performance code for specific contexts. Third, when optimized implementations are applied to other input distributions, the results are often far from optimal. This demonstrates that a “one size fits all” approach would not work here, and if one wants to achieve near optimal performance, they must tune the parameters of their implementation for their specific setting.

In conclusion, we present a compile-time framework which abstracts the values of implementation parameters in favor of their effect of performance. Ultimately, we believe an approach like this one can help tackle the complex implementation space of statistical learning algorithms, which have flexibility in both their algorithmic and systems parameters.